



Automated generation of variable vulnerable architectures

Pierre-Victor Besson

► To cite this version:

Pierre-Victor Besson. Automated generation of variable vulnerable architectures. Cryptography and Security [cs.CR]. CentraleSupélec, 2024. English. NNT : 2024CSUP0019 . tel-05135954

HAL Id: tel-05135954

<https://theses.hal.science/tel-05135954v1>

Submitted on 30 Jun 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE

CENTRALESUPÉLEC

ÉCOLE DOCTORALE N° 601

*Mathématiques, Télécommunications, Informatique, Signal, Systèmes,
Électronique*

Spécialité : *INFO*

Par

Pierre-Victor BESSON

Automated generation of variable vulnerable architectures.

Thèse présentée et soutenue à CentraleSupélec - Rennes, le 22/11/2024 NNT : 2024CSUP0019

Unité de recherche : Laboratoire IRISA - Equipe PIRAT\'; - CentraleSupélec

Rapporteurs avant soutenance :

Jean-Marc ROBERT Professeur, École de technologie supérieure, Montreal, Canada

Guillaume BONFANTE MCF, École des Mines de Nancy · Université de Lorraine

Composition du Jury :

Président : Vincent NICOMETTE

Examineurs : Marion VIDEAU

Dir. de thèse : Valérie VIET TRIEM TONG

Co-dir. de thèse : Gilles GUETTE

Guillaume PIOLLE

Erwan ABGRALL

Professeur INSA Toulouse

Chief of Staff and Chief Scientific Officer, Quarkslab, Paris

Professeur, CentraleSupélec Rennes

MCF, Université de Rennes

Dr. Ingénieur, Thales

Dr. Analyste SOC, EDF-SA

ACKNOWLEDGEMENT

J'aimerais déjà remercier tous mes encadrants de thèse pour leur soutien et enseignements durant ces 4 années. Débuter en confinement fut difficile, mais leur soutien et compétence ont permis d'orienter les travaux de cette thèse dans une direction inattendue au début mais fortement intéressante. Merci Valérie pour sa patience et toutes les opportunités qu'elle m'a amené. Merci Gilles, pour sa disponibilité pour répondre à mes questions plus ou moins pertinentes, et pour m'avoir gentiment expliqué 6 fois comment il faut brancher les câbles pour le TP de réseau. Enfin Guillaume pour son aide pour les aspects formalisme et Erwan pour les aspects techniques, tous deux présents jusqu'au bout malgré leurs nouvelles situations professionnelles. Un grand merci également à Alexandre pour tout le travail technique et la disponibilité, que ce soit sur les honeypots au début ou sur URSID (déso pour le commit à 10h pour CERBERE :p).

Je souhaite également remercier tous les membres de feu l'équipe CIDRE (maintenant PIRAT/SUSHI) pour avoir fait de cet étage de Supelec un endroit agréable à travailler. En particulier Romain, Nathan et Hélène pour l'aventure CERBERE (enfin une publication!!), et toutes les personnes impliquées dans le projet BreizhCTF (Manu, Sébastien...) et qui ont contribué à URSID, mon prototype bricolé en Python qui a finalement fait un sacré chemin depuis. Myriam pour son infinie patience face à mes problèmes de paperasse. Enfin Mathieu et Lionel pour l'élaboration du culte de J.B, avec lecture assidue de ses fines analyses sur le monde contemporain à chaque pause. Je garderais Humour et Management dans ma bibliothèque pour toujours.

D'un point de vue personnel, je souhaite remercier ma famille pour leur soutien durant ces années malgré la distance. Merci également à mes amis pour toutes les distractions et bons moments passés, surtout en temps de COVID. Les gens cools de la scène smash quand j'étais pas encore rincé au jeu. Max pour tous les roguelikes reginald random et les mario karts vin rouge du nouvel an. Canu, pour m'avoir indirectement pété un poumon grâce à des choix judicieux d'upgrades sur Hadès 2 (maintenant fini ta thèse hop hop hop).

Finally, thanks to all my English speaking friends I've met these past years and the fun times spent when timezones allow for it. Pat, your card quality knows no bounds.

TABLE OF CONTENTS

Introduction	7
1 State of the art.	15
1.1 Cyber ranges.	15
1.1.1 Cyber range description and deployment tools.	18
1.2 Attack descriptions	32
1.2.1 Attack graphs	34
1.2.2 Attack step representation.	40
1.2.3 Conclusion of State of the Art.	44
2 Scenario description formalism.	51
2.1 Goals of this formalism.	51
2.2 Detailed formalism	53
2.2.1 Scenario description	53
2.2.2 Attacker progression	58
2.2.3 Conclusion on formalism	62
2.3 Example: formalizing an APT3 inspired attack scenario.	63
2.3.1 Conclusion.	69
3 Procedural refinement.	71
3.1 Procedures and constraints.	72
3.1.1 Problem statement.	72
3.1.2 Architectural constraints.	75
3.2 Backtracking refinement algorithm.	85
3.2.1 Constraint incompatibilities.	85
3.2.2 Procedural refinement.	90
3.2.3 Example: APT-3 inspired scenario.	94
3.3 Conclusion	100

TABLE OF CONTENTS

4	Deployment of vulnerable architectures.	103
4.1	Goal overview.	103
4.2	From URSID to Vagrant/Ansible.	106
4.3	Conclusion.	119
5	Practical experiments using URSID.	123
5.1	CERBERE experiment.	123
5.1.1	Goals, challenges and design choices.	123
5.1.2	Reflections on URSID's use as a CTF generation platform	132
5.2	Experiment results.	134
5.2.1	Course of the experiment.	134
5.2.2	Results analysis.	136
5.2.3	Conclusion of CERBERE experiment	141
5.3	Casinolimit and BreizhCTF.	144
5.3.1	Overview and design constraints.	144
5.3.2	Scenario design and implementation.	146
5.3.3	Event and results.	158
5.4	Conclusion.	163
6	Conclusion	165
	Conclusion	165
6.1	Conclusion	165
6.2	Perspectives	167
	Bibliography	173

INTRODUCTION

IT infrastructures have become a critical component of modern day society, with various organizations, including financial institutions, transportation networks, healthcare facilities and government entities relying heavily on such services. This fact is - sometime painfully - reminded to users whenever a data center encounters issues, or when a faulty update for a Windows security product ends up causing mass outages across the world CISA, *Widespread IT Outage Due to CrowdStrike Update*, URL: <https://www.cisa.gov/news-events/alerts/2024/07/19/widespread-it-outage-due-crowdstrike-update> (visited on 07/26/2024). This heightened importance over the past decades has been met with a logical rise in cyber-related attacks, due to the ever-increasing lucrative ways of monetizing such attacks. For instance, one commonly used method of attackers to leverage their control over a network is to encrypt important data and ask for payment to get it back, a process known as ransomware. The victims are confronted with a stark choice: either try to recover the data or work around the loss on their own - often encountering significant losses in the result - or pay the ransom, which comes with its own share of risks and ethical concerns. Ransom payouts alone crossed the \$1 billion threshold in 2023 Chainalysis, *Ransomware Payments Exceed 1 Billion in 2023, Hitting Record High After 2022 Decline*, URL: <https://www.chainalysis.com/blog/ransomware-2024/> (visited on 07/26/2024) thanks to the wide range of hospitals, city councils or companies being attacked every week.

With the average cost of a data breach being estimated to \$4.45 million in 2023 IBM, *Cost of a Data Breach Report 2023*, URL: <https://www.ibm.com/reports/data-breach> (visited on 07/26/2024), organizations are strongly incentivized to implement robust security practices. These practices aim to reduce the likelihood of successful attacks and mitigate the consequences of such breaches, should they occur. A comprehensive understanding of the methods employed by attackers is paramount to combat these threats. Typically, attacks commence with the acquisition of an entry point into the target architecture by the attackers. This may be achieved through various means ranging from exploiting vulnerabilities in remote access software NIST, *CVE-2019-0708 Detail*, URL: <https://nvd.nist.gov/vuln/detail/cve-2019-0708> (visited on 07/26/2024), to

harvesting user credentials through phishing campaigns, to sending emails containing malicious payloads which provide attackers with control over a user machine NVD, *CVE-2021-40444 Detail*, 201, URL: <https://nvd.nist.gov/vuln/detail/CVE-2021-40444> (visited on 07/26/2024) should they mistakenly activate it. Attackers may then leverage this position in order to perform a variety of actions to improve their control on the architecture. They can learn more about the infrastructure in order to perform future attacks (Discovery), attempt to compromise other accounts (Lateral Movement), or elevate their privileges in order to acquire administrative rights on a machine, or worse, an administrative position within the network.

While each of these actions is theoretically susceptible to detection and reaction by defenders, attackers have at their disposal a range of techniques to mitigate the likelihood of detection, such as compromising detection systems or eradicating traces of their activities. Some groups of attackers known as Advanced Persistent Threats (APTs) also leverage time in their favor by performing their actions over weeks, if not months¹, thereby increasing their stealth. Usually backed by powerful organizations or even states, these attackers can have massive impact on their victims if successful, ranging from huge monetary losses to outright global geopolitical consequences in the case of government actors being affected. For instance, the 2020 SolarWinds supply-chain attack Centre for Cyber Security, *SolarWinds: State-sponsored global software supply chain attack*, URL: <https://www.cfcs.dk/globalassets/cfcs/dokumenter/rapporter/en/CFCS-solarwinds-report-EN.pdf> (visited on 07/26/2024), believed to have originated from the Russian attacker group APT29, resulted in an average financial loss of \$12 million for affected companies, in addition to multiple data breaches involving US government and military entities. Even less sophisticated attacks, such as the distribution of ransomware through malicious or phishing emails² may lead to significant costs for their victims.

From a defender point of view, the stealth of these attackers, in conjunction with their numerous methods of fortifying their control over a network once they have gained access—for example, by implementing backdoors and easier entry points onto a compro-

1. A report by security company FireEye FireEye, *Annual FireEye Mandiant M-Trends Report Reveals Global Statistics and Insights from Hundreds of Diverse Intrusions*, URL: <https://www.mandiant.com/company/press-releases/annual-fireeye-mandiant-m-trends-report-reveals-global-statistics-and-insights-hundreds-diverse-intrusions> (visited on 07/26/2024) puts that number as 24 days in 2020, down from 56 in 2019. Such numbers are however limited to attackers which were successfully detected.

2. Which represent 34% of ransomware based attacks according to a 2024 survey of 5000 cybersecurity leaders Sophos, *The State of Ransomware 2024*, URL: <https://assets.sophos.com/X24WTUEQ/at/9brgj5n44hqvgsp5f5bqcps/sophos-state-of-ransomware-2024-wp.pdf> (visited on 07/26/2024)

mised host—may impede the ability to counteract their actions once infected. Thus, while there is value to be had in such curative measures in order to minimize damages and fix vulnerabilities used by the attackers, these solutions are not able to prevent the issue in the first place. There is therefore a significant incentive to invest in preventive solutions as well. These may include organization wide cyber-security training (mandatory regular password changes, phishing simulations...), keeping systems and software up to date, or good permission management. These measures, if successfully implemented, all limit an attacker's ability to get a foothold on the network or do significant damage inside of it. Even if these are only successful in delaying the attackers, such delays give more opportunities to defenders to successfully detect intrusions. Furthermore, in the case of monetary-driven attackers³, making their work harder leads to worse returns from investments from a business perspective, lowering their incentives to attack one's organization for a long period of time. No defensive measure is perfect and motivated attackers may get in one's network eventually, but combining curative and preventive solutions is the most practical way of reducing the odds of this happening, as well as the damage in case of successful attack.

One valuable option for defender in order to evaluate the vulnerability of their organization is to hire professionals to attack it themselves. These professionals, referred to as "white-hat hackers," "red-teamers," or "pentesters," aim to penetrate the organization's network by exploiting vulnerabilities in a manner similar to an actual attacker, but with non-malicious intentions. Once this penetration test is performed⁴, these groups will provide a detailed report on how they managed to exploit the organization alongside recommendations on how to improve their defense. Consequently, contrary to intuition, a significant component of cyber defense involves the utilization of trained cyberattackers. The skills of these attackers are valuable for testing existing infrastructures and providing defenders with insights into the modus operandi of actual malicious attackers.

However, such attackers have to be trained in order to be effective at their jobs and hone their skills. Theoretical work alone is insufficient; practical experience in exploiting vulnerable architectures is essential for effective learning. This raises the need for practical training platforms, as attacking existing infrastructures is not a viable solution for obvious legal and ethical reasons. Such platforms should facilitate the deployment of architectures in a controlled environment, allowing attackers to develop their pentesting skills without compromising critical systems. This can take the form of Capture-The-Flag cyber compe-

3. Which considering the prevalence of ransomware in today's cyber landscape are quite common.

4. Often abbreviated as a pentest, hence the name pentester.

titions, in which teams of attackers compete to compromise as many challenges (involving cybersecurity tasks) as possible within a given time frame. Another relevant concept is that of cyber ranges, which are virtual environments designed for cybersecurity training. These environments are usually intended with either vulnerabilities (particularly for CTF events) or specific configurations to exploit in mind, in order to practice specific skills for the attackers.

From a defensive standpoint, there exist incentives to deploy an architecture that incorporates a set of vulnerabilities and attacker actions designed to exploit the architecture. We will refer to such a set of actions as an **attack scenario**: the possible set of movements taken by an attacker within the network, as well as how they are intended to use this set of actions in order to reach an objective within the network. This objective can range in scope depending on the scenario, but typically involves the control of a privilege user account within the network, which can then be used to either extract valuable information (data leaks intended to be solved, banking information...) or as a tool to compromise the network as a whole (spreading ransomware being a commonly found way to monetize such attacker endeavors).

The description of these attack scenarios is therefore of interest for vulnerable deployment architecture software, such as the aforementioned cyber ranges. This is because the structure of the scenario is tied to the structure of the architecture itself, as its configuration - network accesses between machines, software installed which may present vulnerabilities, etc.- is what determines the actions available to and expected from the attacker. This description should thus contain both information about these actions and the structure of the architecture itself. The ability to describe an attack scenario and then deploy an architecture related to that scenario provides defenders with significant flexibility. This flexibility enables them to hone a specific skill or match an existing architecture's known vulnerabilities without needing to compromise it directly.

These descriptions have several incentives to balance. On one hand, it has to contain enough information about details of the architectures - or at least, a way to deduce this information from the description -, such as the number of machines, the users to add on those machines, operating systems or software to install. This is because the options available to an attack are strongly correlated to these choices of configuration, offering them different exploits or accesses throughout the network. On the other hand, writing out these descriptions can be a cumbersome task, especially in the case of large networks which can contain dozens if not hundreds of machines. Furthermore, the ability to rewrite and

edit existing scenarios can be very valuable depending on the context of deployment. For instance, once a scenario has been designed and a corresponding vulnerable architecture has been deployed to be played by a set of attackers, the educational value of that specific deployment is lost for those attackers as they already know the answer. Thus, the manner in which this description is achieved can strongly impact how difficult it is to make tweaks to an existing scenario. For instance, how does one change an exploit? Add a new machine? Use similar exploits but modify the order in which they are expected to be performed?

This thesis offers a novel solution to this issue of **scenario reusability**, as part of a new paradigm of attack scenario description and deployment of an architecture vulnerable to that description. Given a vulnerable architecture deployment tool (such as a cyber range), once a scenario has been described once and an architecture corresponding to said description has been deployed and played by attackers, this deployment and description both lose in value for these attackers as they are familiar with the solution to this specific challenge. Ideally, scenario designers are able to quickly and easily tweak descriptions and configurations in order to make new challenges. However, as we will discuss in the state of the art of this thesis, this can often time be a tedious process, involving manual rewriting of machine software or operating systems configurations. The result of this labor is URSID⁵ Pierre-Victor BESSON, *URSID*, URL: <https://gitlab.inria.fr/pirat-public/ursid> (visited on 05/30/2024), an open source vulnerable architecture design and deployment tool intended for research purposes. This thesis, and URSID by extension, aim to provide several variations of a single attack scenario description in order to increase its reusability. It also offers a way to describe a scenario without having to specify configurations of software or operating systems on each machine - a level of description we refer to as high level. We also put interest in being able to describe attacker movement within a scenario: what is their current progress? What are their currently available options in the network? Are they close to their goal? What is the next expected step of their attack?

Applications for this work are multiple from a defender's perspective. Deploying vulnerable architectures is valuable for cybersecurity training - both of professionals and students -, the generation of attacker datasets, and trapping and studying actual attackers in controlled environments in order to learn more about their techniques - a concept known as honeypots⁶.

In order to enhance the reusability of scenarios and facilitate the work of vulnerable

5. Using formalism to Refine attack Scenarios for vulnerable Infrastructure Deployment.

6. In fact, this thesis started with a particular focus on honeypots before branching out to the broader concept of vulnerable architecture deployment as a whole.

architectures deployment, this thesis proposes a novel attack scenario formalism and deployment pipeline. Our first identified scientific problem was to provide the ability for designers to describe their attack scenarios on a high level, freeing them from having to consider specific implementation details. However, this representation still needs to be able eventually to at least infer these configuration choices for later deployment purposes. Our **first contribution** is thus a **new attack scenario formalism**, in which architectures and attacker paths within said architectures can be described at a high level of abstraction - intended for scenario design - or at a lower level - intended for scenario deployment.

As we still aimed for the high level description to be the entry point of deployment of vulnerable architectures, our **second contribution** is a process able to refine a single high level description into several low level ones, an algorithm we dubbed **procedural refinement**. These descriptions all correspond to the same original scenario and share similar structures - i.e attackers are expected to follow similar path regardless of the chosen variation -, but may have significant differences in how these steps are expected to be performed, increasing the variability and reusability of our original description. We also describe our way of converting these low level descriptions into proper vulnerable architecture configuration files fit for deployment, using VagrantHashicorp, *Vagrant*, 2010, URL: <https://www.vagrantup.com/> (visited on 10/23/2023), AnsibleAnsible Inc, *Ansible*, 2012, URL: <https://www.ansible.com/> and VirtualBoxOracle, *VirtualBox*, 2007, URL: <https://www.virtualbox.org/> (visited on 10/02/2023). The combination of high-level scenario formalism, refinement into lower level descriptions and deployment of said descriptions has been implemented in URSID BESSON, *op. cit.*, whose development was a core part of this thesis. A publication that summarizes this entire process, as well as presents URSID, was published at FPS 2023 Besson Pierre-Victor et al., « URSID: Automatically Refining a Single Attack Scenario into Multiple Cyber Range Architectures », *in: Foundations and Practice of Security - FPS 2023*, vol. 14551, 2024, DOI: 10.1007/978-3-031-57537.

Finally, we performed **two experiments** with educational goals using the work of this thesis and its resulting tool URSID. The first, named **CERBERE**, was designed to familiarize university students with the operational processes of professional pentesters and security analysts. The objective of this experiment, as outlined in this thesis, was twofold: first, to leverage our automated scenario variability to present students with diverse challenges, and second, to demonstrate the ability of this thesis work as an attacker dataset generation platform. A paper showcasing the CERBERE experiment as well as making

the dataset publicly available was published at the 2023 IEEE International Conference on Big Data Pierre-Victor Besson et al., « CERBERE: Cybersecurity Exercise for Red and Blue team Entertainment, REproducibility », *in: 2023 IEEE International Conference on Big Data (BigData)*, 2023, pp. 2980–2988, DOI: 10.1109/BigData59044.2023.10386953. The second experiment involved designing a challenge for BreizhCTF, a large scale offline Capture-The-Flag competition with over 600 participants. This challenge, named **Casinolimit**, was another opportunity to use this thesis work in order to acquire attacker data for other research purposes, as well as showcasing our tool’s ability to upscale and deal with the technical challenges that come with such an event. Both of these experiments may be redeployed locally BESSON, *op. cit.*; Alexandre Sanchez, *Casino Limit Challenge*, 2024, URL: <https://gitlab.inria.fr/pirat-public/casinolimit> (visited on 07/23/2024) if one wishes to generate their own dataset with it, as well as use it for educational or entertainment purposes. URSID itself, our new approach to vulnerable architecture deployment which implements all the concepts described in this thesis, is a documented open-source project available online BESSON, *op. cit.*

Chapter 1 offers a comprehensive survey of the current state of the art in vulnerable architecture deployment solutions including cyber ranges, as well as scenario description tools in general. This overview places particular emphasis on each project’s ability to deploy variations of a given scenario and their reusability, and highlights our perceived weaknesses of said state of the art as a result.

Chapter 2 details our attack scenario formalism, which can be used to describe attackers and their path as they move throughout an architecture. This formalism combines the notions of attack graphs and the ATT&CK matrix, both concepts introduced in the state of the art. The main perk of this solution compared to existing ones is its two levels of descriptions: a high-level technical description - named after ATT&CK techniques, which describe attacker actions without considering implementation details -, and a lower level procedural description, which detail the specific exploits or actions an attacker is expected to perform. We also introduce the notion of attacker positions as a way to represent attackers gaining accesses throughout the network, taking inspiration from the works of Aimad Berady, « Comprendre les menaces sophistiquées : Intentions, moyens, manières et connaissances convergentes des adversaires », Theses, CentraleSupélec, Nov. 2022, URL: <https://theses.hal.science/tel-03861429> and Pernelle Mensah, « Generation and Dynamic Update of Attack Graphs in Cloud Providers Infrastructures », Theses, Centrale-Supélec, June 2019, URL: <https://inria.hal.science/tel-02416305> which first for

the purposes of describing existing attacker campaigns and cloud network infrastructures, respectively. We also propose the notion of secrets to show how data acquired as a result of performing attacks can be then reused in the rest of the scenario.

Chapter 3 introduces the concept of procedural refinement. Combined with the notion of architectural constraints, which are ways to represent the conditions a procedure imposes on the network in order to be playable, this refinement allows the conversion of a single high level technical description into several low level procedure ones, which are more fit for deployment. These descriptions all correspond to the same original scenario and share similar structures - i.e attackers are expected to follow similar path regardless of the chosen variation -, but may have significant differences in how these steps are expected to be performed, increasing the variability and reusability of our original description.

Chapter 4 provides a detailed exposition of our methodologies for deploying a vulnerable architecture corresponding to a procedural level scenario, combining virtual machine deployment software stack Vagrant, Ansible and VirtualBox with our notion of architectural constraints.

Finally, Chapter 5 details the two experiments successfully ran using URSID: CERBERE and Casinolimit. As both of them were collaborative work between several researchers, we will put a particular focus on how this thesis work was influential on their design, and how our reusability and flexibility of scenario descriptions helped improve the educational values of these experiments, as well as acquiring valuable attacker data for further research purposes.

STATE OF THE ART.

This thesis addresses the challenge of describing and generating vulnerable architectures for security research applications. These issues are interconnected, as the generation of an architecture - for instance, with the use of virtualization software - necessitates an initial description of the architecture. This section starts with a state of the art on tools dedicated to the generation of vulnerable architectures, commonly referred to as cyber ranges. These tools aim to convert a description of a vulnerable architecture into a playable network. We will put a particular emphasis on how this conversion process is achieved and how the vulnerable architecture descriptions are written. Doing so will raise the question of how to describe attack scenarios in the first place, as the choice of a description language influences the conversion process from scenario to infrastructure. We will thus look into the literature of attack graphs and ways to represent attack scenarios, as they are pertinent to the design of the scenario formalism employed in this thesis. Finally, we will delve into the subtopic of attack step labeling literature, with a particular focus on the MITRE TTP MITRE, *Enterprise Matrix*, 2013, URL: <https://attack.mitre.org/> (visited on 10/24/2023)(Tactics, Techniques and Procedures) nomenclature.

1.1 Cyber ranges.

Cyber ranges are virtual environments designed for cybersecurity training. They are complete information systems similar to those used in production, designed and deployed to test attacker or defender abilities and tools. Cyber range experiments usually involve two different groups. A so-called red team attacks the cyber range, with the goal of compromising the architecture. Concurrently, a blue team assumes the role of defending the cyber range. The specific objectives of the blue team, which may include tracking the attacker's path within the network, slowing down their progress, or gathering and analyzing data to reconstruct their actions, are determined by the experiment's intended purpose.

For instance, European multinational aeronautics enterprise Airbus provides cyber range services to its customers Airbus, *CyberRange*, 2018, URL: <https://www.cyber.airbus.com/cyberrange/> (visited on 01/16/2024), able to model and deploy information systems - based on an existing one or from scratch - in order to reproduce realistic activities or cyberattacks on these deployments. Intended use cases include awareness training of good cybersecurity practices for any staff, honing of skill of dedicated cyber teams, and utilization in pre-production tests in order to have access to a realistic environment.

Cyber range platforms may be deployed with one of several goals in mind:

- **Educational purposes.** The red or blue team may consist of professional white-hat hackers - i.e. hired by organizations to test the weaknesses of their systems - looking to refine their skills, or students in cybersecurity as part of their curriculum.
- **Study attacker behavior.** The cyber range may implement novel defensive techniques and test their effectiveness in a practical but controlled experiment. For instance, one may wish to test the use of honeypots (voluntary vulnerable architectures designed to lure attackers in) and how they may hinder attacker progress Kimberly Ferguson-Walter, « An Empirical Assessment of the Effectiveness of Deception for An Empirical Assessment of the Effectiveness of Deception for Cyber Defense Cyber Defense », PhD thesis, Feb. 2020, DOI: 10.7275/z0rb-ek46; Kimberly Ferguson-Walter et al., « The Tularosa Study: An Experimental Design and Implementation to Quantify the Effectiveness of Cyber Deception », *in*: Jan. 2019. Such data may then be used to reinforce existing defense systems.
- **Gathering attack logs.** A recurring issue in the field of cybersecurity is the lack of available datasets. Uetz *et al.* Rafael Uetz et al., « Reproducible and Adaptable Log Data Generation for Sound Cybersecurity Experiments », *in*: *Annual Computer Security Applications Conference*, ACM, 2021, DOI: 10.1145/3485832.3488020, URL: <https://doi.org/10.1145/3485832.3488020> point out that most work on log data and intrusion detection do not make their underlying datasets available at all, “rendering the reproduction (and thus also extension) of their experiments impossible“. Laundauer *et al.* Max Landauer et al., « Maintainable Log Datasets for Evaluation of Intrusion Detection Systems », *in*: *IEEE Transactions on Dependable and Secure Computing* 20.4 (2023), pp. 3466–3482, DOI: 10.1109/tdsc.2022.3201582, URL: <https://doi.org/10.1109/2Ftdsc.2022.3201582> evaluate the proportion of papers relying on such unavailable datasets to be almost 60%, and observe that among the papers using public datasets, most of them focus on only two of them.

Cyber range based experiments allow for efficient collection of data against reasonably realistic targets. One may argue that such an approach is inherently less realistic than a real attack and that the resulting logs are inherently biased, compared to the gathering of real-life data about network attacks. However, acquiring such data (i.e. actual attack logs from existing architectures) is challenging for several reasons:

- Unlike in red-team exercises, it is hard to gauge when and where a real attack will occur.
- Even if one is able to detect an attack and gather logs from it, such an experiment will be hard to reproduce or tweak.
- Logs related to an attack on an existing architecture are likely to contain sensitive information about its structure or its uses, which means the resulting data will likely have to be anonymized or heavily censored for both privacy and corporate reasons.

Cyber ranges offer an answer to these problems by running the attacks into a more controlled environment.

We will now cover some of these cyber range description and deployment tools. As mentioned in the introduction, the notion of vulnerable architecture is intrinsically tied to the notion of attack scenario. We say that we deploy an attack scenario - a sequence of actions from the hostile party which are able to compromise the system - if we deploy a vulnerable architecture which is weak to this attack scenario, i.e. can be compromise by following that specific sequence of actions. We will focus in particular on cyber range tools ability to deploy different attack scenarios based on a single description and how high or low-level these descriptions are in the first place. We here use the term "low-level" to qualify descriptions which specify every implementation detail of a given cyber range, such as the operating systems, software, or files that must be installed on each machine. In contrast, we will make use of the term "high-level" to describe tools which can deploy a cyber range without having to specify such details, instead only requiring broad information about the network. This may include information such as number of machines, or overall category of exploits to be performed as opposed to specifications of one particular exploit. The level of description - i.e. low vs high level - is of particular interest to this thesis. Initially, it appears that high-level descriptions would be advantageous to cyber range users, as they would facilitate the expeditious and uncomplicated deployment of new architectures. This is due to the fact that users would not be required to specify every detail about the network and the machines that comprise it. This is however a hard task to achieve, as a

low-level description will eventually be needed in the deployment process: the software needs to know what to install on the machines and how many to deploy. The question thus centers on whether this low-level description has to be provided by the scenario designer or whether the software is able to perform a conversion between a high-level and a low-level description, and how flexible - i.e, how much variability in architectures the software can provide based on a single description - this process is.

1.1.1 Cyber range description and deployment tools.

Cyber range research has seen a noticeable increase in recent years, thanks in part to the increasing importance of cybersecurity in all aspects of society and improvements in emulation software. Some cyber range deployment tools are fully available online with open source code and implementation Z. Cliffe Schreuders et al., « Security Scenario Generator (SecGen): A Framework for Generating Randomly Vulnerable Rich-scenario VMs for Learning Computer Security and Hosting CTF Events », *in: 2017 USENIX Workshop on Advances in Security Education (ASE 17)*, Vancouver, BC: USENIX Association, Aug. 2017, URL: <https://www.usenix.org/conference/ase17/workshop-program/presentation/schreuders>; Uetz et al., *op. cit.*; Max Landauer et al., « Have it Your Way: Generating Customized Log Datasets With a Model-Driven Simulation Testbed », *in: IEEE Transactions on Reliability* 70.1 (2021), pp. 402–415, DOI: 10.1109/TR.2020.3031317; Jaromír, *NASimEmu: Network Attack Simulator and Emulator*, 2023, URL: <https://github.com/jaromiru/NASimEmu/>, while others only provide information about their inner workings Sridhar Venkatesan et al., « VulnerVAN: A Vulnerable Network Generation Tool », *in: MILCOM 2019 - 2019 IEEE Military Communications Conference (MILCOM)*, 2019, pp. 1–6, DOI: 10.1109/MILCOM47813.2019.9021013; Muhammad Mudassar Yamin and Basel Katt, « Modeling and executing cyber security exercise scenarios in cyber ranges », *in: Computers & Security* 116 (2022), p. 102635, ISSN: 0167-4048, DOI: <https://doi.org/10.1016/j.cose.2022.102635>, URL: <https://www.sciencedirect.com/science/article/pii/S0167404822000347>.

There are two main aspects to the description of cyber range architectures. On one hand, a cyber range’s goal is to deploy a system and network architecture, and one may thus expect its description to be a specification of this architecture itself: number and type of machines, network configurations, remotely accessible entry points, etc. On the other hand, a cyber range also corresponds to an attack scenario it is vulnerable to, which may also be described. That is to say, some cyber range deployment tools

double as attack scenario description tools, and are able to convert these descriptions into vulnerable architectures. Existing description languages and tools deal with one Landauer et al., *op. cit.*; Jaromír, *op. cit.*; Gabriele Costa, Enrico Russo, and Alessandro Armando, « Automating the Generation of Cyber Range Virtual Scenarios with VSDL », *in: Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* 13.4 (Dec. 2022), pp. 61–80, DOI: 10.58346/jowua.2022.i4.004, URL: <http://dx.doi.org/10.58346/JOWUA.2022.I4.004>; Uetz et al., *op. cit.*; Cuong Pham et al., « CyRIS: a cyber range instantiation system for facilitating security training », *in: Dec. 2016*, pp. 251–258, DOI: 10.1145/3011077.3011087 or both Venkatesan et al., *op. cit.*; Schreuders et al., *op. cit.* of these aspects. Cyber range specifications may also include instructions of technical details on how these descriptions are converted into proper architectures by using deployment tools Venkatesan et al., *op. cit.*; Schreuders et al., *op. cit.*

For educational purposes, Schreuders *et al. idem*, « Security Scenario Generator (SecGen): A Framework for Generating Randomly Vulnerable Rich-scenario VMs for Learning Computer Security and Hosting CTF Events » propose SecGen, a reusable Capture-The-Flag platform generation intended for the training of university students. In this context, a Capture-The-Flag event refers to a cyber-security exercise or contest in which teams of attackers have to penetrate a system in order to extract data - the flag - in order to prove their achievements. While not explicitly presenting itself as a cyber range, it is similar enough in function to be mentioned here. SecGen is divided into modules which lets its user specify details about the infrastructure to be deployed through XML descriptions. This includes operating systems, vulnerabilities, services, software (through the "Utility" module), or networks. For instance, one may wish to deploy a machine that runs on a Debian type distribution and contains a privilege escalation exploit, making use of the operating system and vulnerability modules. An example of such a scenario description is available in figure 1.1. Users may thus provide their own modules for each category, adding information such as difficulty, requirements and conflicts with other modules, or software to be installed for the module to work. This allows for higher level scenario description: a user is for instance able to specify that they want a machine vulnerable to any kind of remote privilege escalation (by taking a module at random that satisfies this category), as opposed to having to specify which type of escalation they want directly.

As for deployment, a scenario is first described as a combination of modules, detailing information such as operating systems or network configuration. Then SecGen assesses the compatibility of these modules among themselves, generates configuration files (using

```
<?xml version="1.0"?>

<scenario xmlns="http://www.github/cliffe/SecGen/scenario"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.github/cliffe/SecGen/scenario">

  <system>
    <system_name>random_escalatable_server</system_name>
    <base distro="Debian" type="desktop" name="KDE"/>
    <vulnerability access="remote" privilege="user_rwx"/>
    <vulnerability access="local" privilege="root_rwx"/>
  </system>
</scenario>
```

Figure 1.1 – Example of a scenario description in SecGen, specifying a single machine with a random remote privilege escalation. Taken from https://github.com/cliffe/SecGen/blob/master/scenarios/examples/random_remote_escalation.xml

Puppet Puppet Labs Limited, *Puppet*, 2005, URL: <https://www.puppet.com/> (visited on 10/23/2023) for system configuration and Vagrant Hashicorp, *op. cit.* for virtual machine configuration) and finally instantiates the result. SecGen’s module system offers a high degree of flexibility when designing scenarios. In particular, the idea of associating them with conditions and requirements or conflicts is one that will be seen again throughout the work of this thesis, albeit with a different execution. Furthermore, even with a fixed set of modules, scenarios offer some level of re-usability to them as passwords or flags are generated dynamically with each instance. This is of particular interest for their educational purposes, as this limits the ability of students to reuse each other’s results and bypass the learning process. Additionally, SecGen provides higher-level descriptions for select system configurations. For instance, it enables users to request a privilege escalation vulnerability without the necessity of selecting a specific vulnerability, thereby enhancing flexibility and reusability. However, outside these specific cases scenario descriptions remain fairly low-level, as modules themselves are associated with precise descriptions of the architecture they required, i.e. specific OS/software to be installed. For instance, the example available in figure 1.1, while offering variability in the type of privilege escalation, still has to specify that the machine has to run on a Debian operating system. The lack of formalized way to describe an attack scenario or the path of an attacker in SecGen is also worth noting. While this completely makes sense for their use case, which is very practically oriented toward the

deployment of Capture-The-Flag architectures, it may be interesting for research purposes to showcase the overall scenario and how an attacker evolves through it, two aspects not covered by this work.

Relatedly, Pham et al. Pham et al., *op. cit.* propose CyRIS, a tool that automatically deploys cyber ranges based on YAML descriptions for education and training purposes. CyRIS also provides options to emulate attacks or malware on the resulting architectures, which will then be recognizable in the logs. Scenarios are not here described directly. Instead, the architecture is described on a low-level by providing information about software, operating systems, accounts or files to modify and install on the machines. These modifications on the machine may lead to exploits and procedures being available to the attacker, providing them with a possible attack scenario to perform. An extract of a scenario description covering one machine by CyRIS is available in figure 1.2. It directly specifies information such as the packages (nmap, wireshark, etc.), files (flag.txt), accounts (daniel), and IP (192.168.122.50) of the machine. Due to the lower level of this description, attack scenarios are here akin to a sequence of attack procedures, i.e. specific exploits and actions the attacker is expected to strictly follow. In comparison to SecGen (which started development a year later), CyRIS lacks the flexibility of the module system: adding a new vulnerability requires manually writing which files, software or OS have to be added on the machine. It does however allow for attack and malware emulation on the instances, which may then be seen in the logs. Example of such attacks include an SSH password brute-force, and a little malware-emulating program which listens to ports and performs simple calculations. Project development also seems to have slowed down around 2020, at least publicly Pham et al., *CyRIS*, 2016, URL: <https://github.com/crond-jaist/cyris>. However, during the writing of this thesis in January 2024 the repository was updated in order to indicate that the project was migrating to a new repository and under new management, due to the original project having been shut down in 2021.

For log gathering purposes, Uetz et al. Uetz et al., *op. cit.* propose SOCBED, an open-source Uetz et al., *SOCBED*, 2021, URL: <https://github.com/fkie-cad/socbed>, virtual cyber range deployment infrastructure which aims to automate attacks in order to create reusable datasets for research purposes. Using Ansible Inc, *op. cit.*, Packer Hashicorp, *Packer*, 2013, URL: <https://www.packer.io/> (visited on 07/02/2023) and VirtualBox Oracle, *op. cit.*, SOCBED generates a small vulnerable network with simulated user activity and automated log collection, and provides tools to attack it in order to extract logs from said attack. A visual representation of their deployment architecture is

```
– guest_settings:
– id: desktop
  ip_addr: 192.168.122.50
  basevm_host: host_1
  basevm_config_file: /home/cyuser/images/basevm_desktop.xml
  basevm_type: kvm
  tasks:
  – add_account:
    – account: daniel
      passwd: danielpass
      full_name: Daniel Radcliffe
    – account: root
      new_passwd: abcd1234
  – install_package:
    – package_manager: yum
      name: wireshark
      version: 1.8.10
    – package_manager: yum
      name: GeoIP
    – package_manager: yum
      name: nmap
  – emulate_attack:
    – attack_type: ssh_attack
      target_account: daniel
      attempt_number: 200
      attack_time: 2017-11-18
  – emulate_malware:
    – name: daemon
      mode: dummy_calculation
      cpu_utilization: 40
  – copy_content:
    – src: /home/cyuser/database/penetration_testing
      dst: /bin/cyberrange
    – src: /home/cyuser/database/flag.txt
      dst: /root
  – execute_program:
    – program: /bin/cyberrange/database/penetration_testing/install_pip.sh
      interpreter: bash
    – program: /bin/cyberrange/database/penetration_testing/prepare.sh
      interpreter: bash
      args: arguments
      execute_time: after_clone
  – firewall_rules:
    – rule: /home/cyuser/cyris-development/database/firewall_ruleset_forwarding
    – rule: /home/cyuser/cyris-development/database/firewall_ruleset_inputoutput
```

Figure 1.2 – Extract of a scenario defined by CyRIS. Taken from <https://github.com/crond-jaist/cyris/blob/master/examples/full.yml>

available in figure 1.3. The example scenario mentioned and tested in Uetz *et al.* Uetz *et al.*, *op. cit.* publication is as follows. First, the attacker is meant to exploit a SQL vulnerability on a web server, in order to acquire information about the employees of a fake company created for the experiment. Using this information, the attacker crafts spear-phishing emails targeted at the employees, which leads to them acquiring remote http command and control abilities. The attacker then collects screenshots of the machine before performing a privilege escalation in order to acquire administrative rights. Using those newfound privileges, they perform lateral movement to another machine on the network which appears to have important documents. Finally, they install and launch a backdoor on this machine for future access. This multistep cyber-espionage type scenario was selected by Uetz *et al.* for its tendency to elude detection (compared to attacks such as ransomware, which have very obvious consequences). It also spans across a lot of different attacker tactics, such as privilege escalations (acquiring a session on an account with administrative rights), lateral movements (moving from one machine to another) or credential access and gathering (rummaging through a machine to acquire passwords or keys). SOCBED also includes a complete logging system able to detect the attacks, by having each deployed machine regularly send its relevant logs to a dedicated login server.

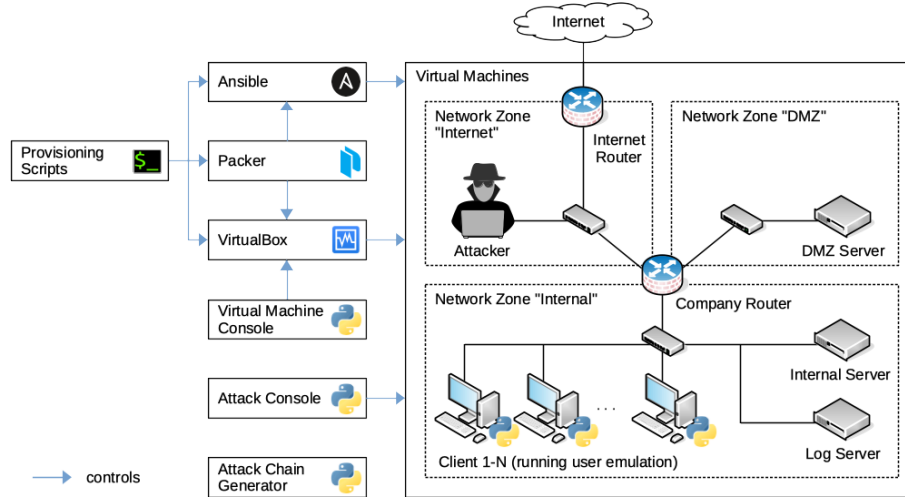


Figure 1.3 – Architecture of SOCBED. Taken directly from the work of Uetz *et al.* Uetz *et al.*, *SOCBED*, 2021, URL: <https://github.com/fkie-cad/socbed>.

However, SOCBED aims to generate logs that are as similar to each other as possible in each instance that is deployed. This is achieved by purposefully providing very little variation in the scenario that is deployed. For instance, installed software versions are always the same. Obstacles mentioned by Uetz *et al.* for the replicability of their experiment

include random user activity and operating system automatically downloading updates. While SOCBED is an effective tool for the purpose of attack log gathering, its design limits its applicability for generalized and varied cyber range deployment.

In a similar vein, Laundauer et al. Landauer et al., *op. cit.* propose Kyoushi, which simulates an enterprise IT network with the goal of acquiring attack logs. In comparison to SOCBED, Kyoushi places a greater emphasis on its ability to modify the entry scenario, as one of the author’s critiques over existing testbeds is their inability to be edited without being manually rewritten. Kyoushi is able to randomize usernames and passwords when deploying a scenario, as well as usernames and IPs when automatically attacking said scenario. These deployments do not however differ significantly in terms of attack procedures: this requires modifying the attack scenario itself. Nonetheless, Kyoushi’s modular structure makes it easy to tweak the scenario - for instance by introducing new vulnerabilities - by modifying those components only while leaving everything else untouched. Comparison with SecGen Schreuders et al., *op. cit.* is less relevant as the tools have different purposes, with SecGen being centered on education while Kyoushi aims to gather attack logs. As such, tools such as SecGen’s ability to provide hints or Kyoushi’s extended logging system are not necessarily relevant for the other. We may however note that SecGen appears to have deeper scenario customization abilities, while Kyoushi provides tools to be deployed on an OVH OVH, *OVHCloud*, 1999, URL: <https://www.ovhcloud.com/fr/> (visited on 08/02/2024) cloud platform. The selection of one tool over another should be informed by the specific requirements of the given use case. In the case of SOCBED against Kyoushi, one may recommend to use SOCBED if there is a need for the logs to be easily reproducible or to compare results against an existing experiment, while Kyoushi is more suitable for deploying diverse scenarios.

Another aspect of cyber ranges for log gathering is their utility for AI training. Indeed, AI agents tend to require substantial amounts of data for training, necessitating the development of methods for generating additional datasets as needed. Janisch et al. Jaromír Janisch, Tomáš Pevný, and Viliam Lisý, *NASimEmu: Network Attack Simulator and Emulator for Training Agents Generalizing to Novel Scenarios*, 2023, arXiv: 2305.17246 [cs.CR] introduce NASimEmu, a network attack simulation and emulation tool designed for the training of autonomous agents. In this context, network attack simulation involves the representation of a scenario as a finite-state machine traversable by the AI, as opposed to the deployment of the architecture itself. This approach, while expediting the training of AI, comes at the expense of realism. Based on NASim Jonathon Schwartz, *Network Attack*

Name	Goal	Scenario description.	Notes on replayability of a single description.
SecGen	Educational	List of software, OS, networking configuration, vulnerabilities...	Randomized passwords, sometime vulnerabilities.
CyRIS	Educational	List of software, OS, vulnerabilities...	Randomized passwords and usernames.
SOCBED	Log gathering	Fixed scenario.	Not relevant.
Kyoushi	Log gathering	Fixed infrastructure, choose attacker behavior.	Randomized passwords and usernames. Can simulate user behavior.
NASimEmu	AI training	Choose software, OS, networking...	Can pick a randomized selection of those parameters.

Table 1.1 – Summary of cyber range deployment tools seen so far.

Simulator, 2018, URL: <https://github.com/Jjschwartz/NetworkAttackSimulator>, a previous open source tool which only dealt with simulation, NASimEmu is able to deploy architectures based on YAML descriptions of the services, OS, users and networking configuration of the machines in the scenario. NASimEmu also provides a method for generating a scenario with a random combination of those parameters, taking as input the size of the network and the number of exploits desired. This is done in order to train general agents able to tackle a wider range of challenges. Finally, a third dynamic mode allows for partial randomization, where the number of hosts in a subnet and their configuration is randomized, while the list of possible parameters (OS, services, exploits...) and network topology stay fixed. Unfortunately, the paper does not go into details about how this randomization is performed, but the overall idea provides interesting re-usability and variability to scenario designs. The examples available on the project’s Git repository Jaromír, *op. cit.* do not appear to make use of the randomization abilities, and are as such fairly low-level descriptions. CybORG Callum Baillie et al., *CybORG: An Autonomous Cyber Operations Research Gym*, 2020, arXiv: 2002.10667 [cs.CR], a cybersecurity research environment for training autonomous agents, is also supposedly able to deploy scenarios based on YAML descriptions. However, according to the previously mentioned paper Janisch, Pevný, and Lisý, *op. cit.*, this part of their work was discontinued and never actually came to fruition.

Table 1.1 offers a summary of the cyber range deployment tools presented so far. We

may note that the notion of scenario description up to this point has been fairly unexplored, and is for these tools equivalent to the notion of low level architecture description. Users of these tools are expected to come up with a list of software, OS or vulnerabilities to describe their architectures and then expect the result to be a playable and interesting scenario for the attacker. This is as opposed to writing down an attack scenario, which would then be converted into information about the architecture itself.

Costa *et al.* Costa, Russo, and Armando, *op. cit.* propose VSDL (Virtual Scenario Description Language), which aims to ease the automation of cyber range deployment in order to increase their variability. Expanding on their previous work on SDL Enrico Russo, Gabriele Costa, and Alessandro Armando, « Scenario Design and Validation for Next Generation Cyber Ranges », *in: 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, 2018, pp. 1–4, DOI: 10.1109/NCA.2018.8548324, a scenario is defined as a name and a sequence of elements, which may be either nodes (i.e. machines) or networks, alongside their conditions. For instance, a node may be defined as a Phone node, with an associated mobile flavor, (useful for hardware specifications), a CPU slower than 2 GHz, and with an OS of Android 21 OR Android 19. An example of such node configurations is available in figure 1.4. VSDL is also able to specify firewall configurations or software to be installed, and tweak the state of the scenario over time - for instance, by adding a machine to the network after 40 minutes. Specific CVEs may also be installed, and are translated as Software and Operating System conditions. These node definitions are then automatically converted into logic statements (Satisfiability Modulo Theories Clark Barrett et al., « Satisfiability Modulo Theories », *in: Handbook of Satisfiability, Second Edition*, ed. by Armin Biere et al., vol. 336, Frontiers in Artificial Intelligence and Applications, IOS Press, Feb. 2021, chap. 33, pp. 825–885, URL: <http://theory.stanford.edu/~barrett/pubs/BSST21.pdf>), an example of which can be seen in figure 1.5, derived from the nodes defined in 1.4. VSDL then uses a logic solver in order to check whether these logic statements, and thus the resulting architecture, is consistent in terms of hardware or network conditions. If the architecture is not deemed consistent, an error message is returned.

The idea of associating CVEs with Software and Operating System conditions in order to populate the cyber range with vulnerabilities is sound. One may however wonder how this work deals with CVEs that require more specific conditions in order to be exploited, such as tweaking configuration files or launching services. The scenario descriptions offered by VSDL also remain fairly low-level, with the user having to specify the hardware,

```

1 node Phone {
2   flavour is mobile;
3   not (disk is larger than 8GB);
4   not (cpu is faster than 2GHz);
5   (OS is Android-21) or (OS is Android-19);
6 }
7 node ApacheS {
8   flavour is server;
9   disk is larger than 200GB;
10  cpu is faster than 8GHz;
11  OS is Debian-8;
12  mounts software apache2;
13  mounts software php5;
14  mounts software dvwa-setup.sh;
15 }

```

Figure 1.4 – Example of node definitions in VSDL Gabriele Costa, Enrico Russo, and Alessandro Armando, « Automating the Generation of Cyber Range Virtual Scenarios with VSDL », in: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* 13.4 (Dec. 2022), pp. 61–80, DOI: 10.58346/jowua.2022.i4.004, URL: <http://dx.doi.org/10.58346/JOWUA.2022.I4.004>, one for a phone and one for an apache server.

```

1 ; Scenario elements
2 (declare-fun Phone () Int)
3 (declare-fun ApacheS () Int)
4 (declare-fun Laboratory () Int)
5 ; ...
6 ; Hardware constraints: Phone
7 (assert (forall ((u Int)) (and (< (node.cpu u Phone) 16192) (< (node.disk u Phone) 32768))))
8 (assert (forall ((u Int)) (and (>= (node.cpu u Phone) 512) (>= (node.disk u Phone) 2048))))
9 (assert (forall ((u Int)) (< (node.disk u Phone) 8192)))
10 (assert (forall ((u Int)) (< (node.cpu u Phone) 2048)))
11 ; ...
12 ; Network constraints: Laboratory
13 (assert (forall ((u Int) (n Int))
14   (or
15     (and
16       (<= (network.node.address u n Laboratory) 134744065)
17       (>= (network.node.address u n Laboratory) 134744128)
18     )
19     (= (network.node.address u n Laboratory) 0)
20   )))

```

Figure 1.5 – Excerpt of SMT logic statements in VSDL Gabriele Costa, Enrico Russo, and Alessandro Armando, « Automating the Generation of Cyber Range Virtual Scenarios with VSDL », in: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* 13.4 (Dec. 2022), pp. 61–80, DOI: 10.58346/jowua.2022.i4.004, URL: <http://dx.doi.org/10.58346/JOWUA.2022.I4.004>

network, software and OS of the entire scenario. The idea of checking for inconsistencies in a description is nonetheless particularly relevant for the goal of this thesis, and our own approach to this issue will be expounded upon in subsequent sections.

The work of Yamin *et al.* Yamin and Katt, *op. cit.* is of particular interest and relevance to the subject of this thesis. The authors explore the question of cyber range generation from scenario design to proper architecture deployment. They define scenarios as a combination of network topology, machines, vulnerabilities, and abilities of attacker/defender, all of which can be configured. For instance, machine configurations specify the OS, the name, the related subnet and a SSH key to provision it with. Vulnerabilities implemented via their tool can be injected into a machine by specifying their name and parameters, such as the name of a .exe file to run or a the value of a password to change. These parameters appear to be specific to the type of vulnerability. Datalog Jay McCarthy, *Datalog*, 1977, URL: <https://docs.racket-lang.org/datalog/index.html> (visited on 11/22/2023), a programming language based on declarative logic, is employed to specify these scenario conditions, facilitating the examination of potential attacker paths within the scenario. An interesting point of their modelization is the fact that vulnerabilities are each mapped to steps of a typical attack model known as the Cyber Kill Chain lockheedmartin, *The Cyber Kill Chain*, 2011, URL: <https://honeyscore.shodan.io/> (visited on 08/30/2023), making it possible to check the scope of the current scenario. Each vulnerability thus provides information about which part of a kill chain it may be associated with. For instance, a privilege escalation vulnerability is likely to be associated with the "Exploitation" step. This makes it theoretically possible to design a scenario by specifying Kill Chain steps to follow as opposed to specific vulnerabilities and let the software figure out the details, but the paper does not seem to have considered this possibility, which is one of the points this thesis will try to address. The work of Yamin et al. seems to have been successfully tested as part of cybersecurity exercises and events, but does not appear to have any public release.

Finally, while it unfortunately lacks any publicly available implementation as far as the author of this thesis is aware, the work of Venkatesan *et al.* with VulnervAN Venkatesan et al., *op. cit.* is of particular relevance to the subject of this thesis. VulnervAN is a vulnerable network generation tool that takes a network description and attack steps as inputs and deploys virtual machines vulnerable to these attack steps as outputs. Networks are described as a combination of segments, such as the Internet, the DMZ, or an enterprise network. For each segment, users specify the type and number of machines (Linux, Windows,

etc.) alongside the type of services run on those machines (DNS, email, etc.). Attack paths, in turn, are modelled as a sequence of steps, with each step being associated with a MITRE ATT&CK technique. The MITRE Tactics, Techniques, and Procedures nomenclature will be detailed in a later section. Attack steps consist of preconditions, pre-execution steps, an action, post-conditions, and post-execution steps. Attack steps may also take arguments to specify information about their deployment. For instance, they may take arguments denoting which specific payload to use as part of a phishing attack. Alternatively, they may take arguments indicating that two steps take place on the same machine. For instance, an email user execution attack may follow a spear phishing link attack on the same machine. Once a network and attack path have been provided by the user, VulnerVAN attempts to correlate both of those inputs together by calculating all possible attack paths that may correspond to the given attack sequence within the network. It then generates virtual machines corresponding to all the implicated parts of the networks, alongside instructions on how to populate them (software to install, services to launch...). Finally, upon generating the architecture, VulnerVAN provides detailed instructions on how to attack the resulting network. Figure 1.7 showcases an attack-step definition, related to a spearphishing attack. This description specifies information about three machines: an Attacker, an EmailServer and a Target. For instance, the Target must be of type *workstation_type* and run on a Windows 7 operating system, while the EmailServer must be in the network region *internet*. This description is also able to specify network access between the machines: for instance, the Attacker machine must have access to port 25 of EmailServer. Finally, the execution of this attack step leads to a post condition stating that machine Target must have received a spearphishing email. These pre and post conditions are represented by boolean predicates, which are then converted into Prolog rules and computed in order to verify whether the resulting architecture is possible, i.e. that no incompatibility between the predicates emerged. It appears that these are defined on a lower level than MITRE techniques, for instance specifying that the target OS must be Windows 7, but is still pretty flexible. Overall, VulnerVAN appears to offer a satisfactory degree of variety in the architectures it is able to output, granted that the attack steps are implemented in the first place.

Summary and limits of cyber range deployment state of the art This section demonstrated that the extant literature on cyber range deployment addresses numerous facets of cyber range utilization, ranging from its application in generating datasets for

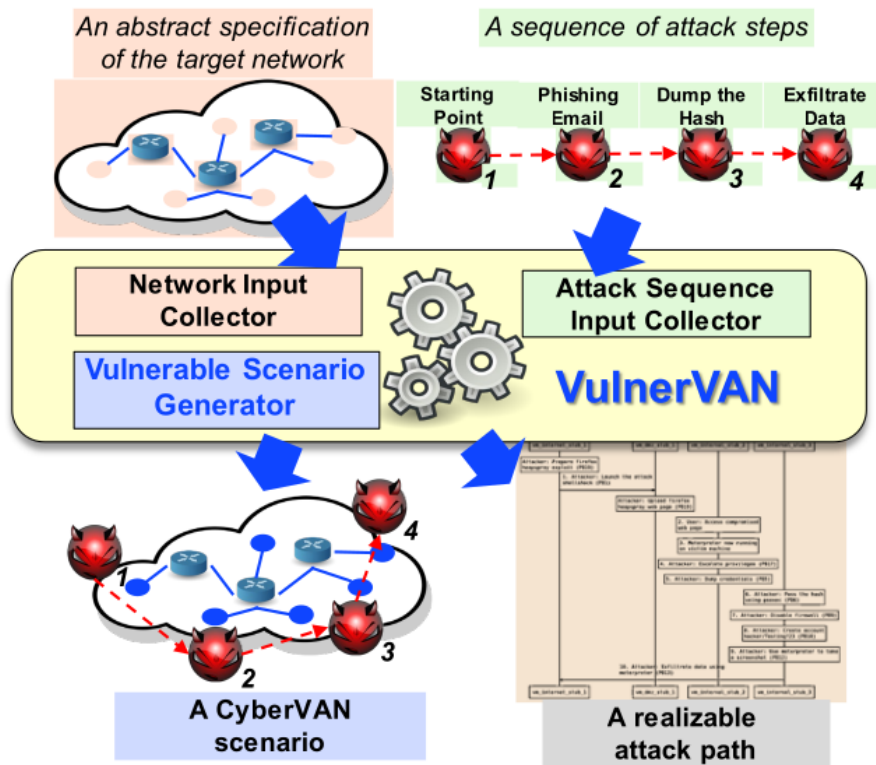


Figure 1.6 – High-level architecture of VulnerVAN. Taken directly from the work of Venkatesan et al. Sridhar Venkatesan et al., « VulnerVAN: A Vulnerable Network Generation Tool », in: *MILCOM 2019 - 2019 IEEE Military Communications Conference (MILCOM)*, 2019, pp. 1–6, DOI: 10.1109/MILCOM47813.2019.9021013.

further research to its role as a training platform for students. These tools mostly rely on low level description of the resulting architecture (often in XML or YAML), where the user has to describe the configuration (i.e. OS, software, services, vulnerabilities, etc.) of each machine manually. Consequently, these descriptions are characterized by their static nature and limited reusability, necessitating manual rewriting of configuration files for each intended adjustment to the deployed architecture. This has implications for the reusability of cyber ranges, as modifying the architecture necessitates the manual creation of new configuration files. Some of the work presented here does offer solutions to this problem: SecGen’s Schreuders et al., *op. cit.* module system makes it easier to add existing vulnerabilities or software to new scenario descriptions, NASimEmu Janisch, Pevný, and Lisý, *op. cit.* is able to randomize a given subset of machine configuration, and VulnerVAN Venkatesan et al., *op. cit.* generates networks based on attack step



Figure 1.7 – Attack-step definition for Spear Phishing Link technique in VulnervAN. Taken directly from the work of Venkatesan et al. Sridhar Venkatesan et al., « VulnervAN: A Vulnerable Network Generation Tool », in: *MILCOM 2019 - 2019 IEEE Military Communications Conference (MILCOM)*, 2019, pp. 1–6, DOI: 10.1109/MILCOM47813.2019.9021013

descriptions, meaning changing the architecture can be done by just tweaking said attack step. The work of Yamin et al. Yamin and Katt, *op. cit.* is arguably the closest to the intended goal of this thesis, with their ability to associate vulnerabilities with Cyber Kill Chain steps (a higher level description) and compute resulting architectures while making sure they are consistent. However, the description does remain low level as it tags vulnerabilities with Cyber Kill Chain steps, as opposed to designing the scenario with those steps as a basis. Furthermore, having been designed in 2011, the Cyber Kill Chain has some weaknesses¹ and outdated aspects in its abilities to represent scenarios

1. We will expand on that point in a later section of the state of the art.

Chrissy Kidd, *Cyber Kill Chains Explained: Phases, Pros/Cons and Security Tactics*, 2022, URL: https://www.splunk.com/en_us/blog/learn/cyber-kill-chains.html (visited on 03/04/2024).

Table 1.8 offers a summary of the reviewed cyber range state of the art. In this table, *Scenario variability* refers to how many different types of attack scenarios and architectures vulnerable to these scenarios can be deployed. *Manual work required to tweak scenario* refers to how much configuration files and code the scenario designer has to change when they wish to deploy a different scenario. While this rating is ultimately subjective, we consider that a lot of work is required if the designer has to specify each machine configuration by hand, while some work is required if shortcuts are offered to the designer (such as pre-configured vulnerabilities or ability to randomize some configuration), but manual input of some machine configuration - such as its operating system or software - is still required.

We believe the state of the art main limitation to be that while these tools are very efficient at describing and adapting different scenarios, their starting point remains a low-level configuration file, with one scenario description achieving one architecture. VulnerVAN procedures are fixed within a description, NASimEmu offers some randomization, but it remains unclear how it is achieved. We wished to push this idea of abstracting scenarios even further in order to easily deploy different instances corresponding to the same high-level description of a given scenario. So far, the question of cyber range representation has been approached from a pragmatic perspective, with the explicit objective of modeling architectures for deployment. However, there has been limited emphasis on the representation of the attack scenario itself. How can one model the actions and trajectory of an attacker within a network, along with the pertinent components of that network? Part of this thesis motivations being our ability to deploy an architecture based on a scenario description, this question was thus at the core of the early designing stages. The next section will deal with the literature’s answers to that question, by introducing a concept known as *attack graphs*, the many ways such a graph can be designed, and delve in particular in a few works that particularly influenced this thesis’ scenario modelization.

1.2 Attack descriptions

In the event that the objective is to deploy a vulnerable architecture based on a scenario description, it is evident that a method for describing said scenario is necessary.

SecGenZ. Cliffe Schreuders et al., « Security Scenario Generator (SecGen): A Framework for Generating Randomly Vulnerable Rich-scenario		CyRIS
	KyoushiMax Landauer et al., « Have i	
	NASimEmuJaromír Jani	
VSDLGabriele Costa, Enrico Russo, and Alessandro Armando, « Automating the Generation of Cyber R		
Yamin <i>et al</i> Muhammad Mudassar Yamin and Basel Katt, « Modeling and executing cyber secu		
VulnerVANSridhar Venkatesan et al., « V		
		1.2. Attack descriptions

Figure 1.8 – Summary of cyber range deployment tools presented in this thesis.

So far, while some of the cyber range deployment platforms mentioned deal with the representation of a scenario, they mostly see it in a very practical way via describing the physical components (machines, software, vulnerabilities) which have to be deployed. This is as opposed to a more formal approach, in which a scenario would first be represented as a mathematical object before being then converted into a more practical description. The benefits of such a formal structure would be a) a clear template to follow for the description of any attack scenario b) ability to perform mathematical operations and algorithms on said structure. For example, graph traversal algorithms could be employed for graph representations to ensure that a given scenario contains winning paths.

One area of the literature dedicated to this problem is the one of attack graphs. This section will delve into the definition of attack graphs, the various representations that can be used when designing one, and the manner in which they can be applied to the objective of this thesis.

1.2.1 Attack graphs

Attack graphs are formal structures that aim to represent one or more possible attacks on an architecture using nodes and edges between those nodes. They differ in the literature based on how they decide to represent the architecture, which depends on their use case: someone attempting to model an attack scenario in order to deploy a network later may not have the same needs for their modeling as someone more interested in the attack itself. The literature identifies 3 overall categories of attack graphs:

- **State-based model** (such as Cynthia Phillips and Laura Painton Swiler, « A graph-based system for network-vulnerability analysis », *in: Proceedings of the 1998 Workshop on New Security Paradigms*, NSPW '98, Charlottesville, Virginia, USA: Association for Computing Machinery, 1998, pp. 71–79, ISBN: 1581131682, DOI: 10.1145/310889.310919, URL: <https://doi.org/10.1145/310889.310919>) in which a node represents the system state after the occurrence of an event and the edges represent the transitions between these states after said events happen. "System state" here depends on the choice of implementation, but can be seen as a series of boolean pertaining to the current attack. For instance, in a simplified example, an attacker may go from a state "machine A is not compromised and has not been harvested for credentials" to "machine A is compromised and has not been harvested for credentials" to "machine A is compromised and has been harvested for

credentials”. The number of different states depends on the choices of representations, but this choice is at the risk of exponential complexity Xinming Ou, Wayne F. Boyer, and Miles A. McQueen, « A Scalable Approach to Attack Graph Generation », *in: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS '06*, Alexandria, Virginia, USA: Association for Computing Machinery, 2006, pp. 336–345, ISBN: 1595935185, DOI: 10.1145/1180405.1180446, URL: <https://doi.org/10.1145/1180405.1180446>. A graphical example of a state-based attack graph is available in figure 1.9. This example showcases a scenario in which the attacker aims to get access to machine M. They may do so either by acquiring a root session on machine M using their physical access to this machine (left side of the graph), or going through another machine B (right side of the graph).

- **Vulnerability-based model** (such as Kyle Ingols, Richard Lippmann, and Keith Piwowarski, « Practical Attack Graph Generation for Network Defense », *in: 2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006, pp. 121–130, DOI: 10.1109/ACSAC.2006.39), in which the graph nodes and edges illustrate the conditions, effects and interdependence of attacks and exploits in a system. Figure 1.10 shows a graphical example of a vulnerability-based model, representing the network as a series of devices, conditions and vulnerabilities to reach these devices.
- **Host-based model** (such as P. Ammann et al., « A host-based approach to network attack chaining analysis », *in: 21st Annual Computer Security Applications Conference (ACSAC 05)*, 2005, 10 pp.–84, DOI: 10.1109/CSAC.2005.6), in which nodes represent an architecture device and edges the accesses available between these devices. For instance, nodes may represent machines and edges the exploits necessary to go from one machine to the other.

Among these, we judged host-based model to be particularly relevant for the objective of modeling and deploying vulnerable architectures. Indeed, information about devices can be more easily converted into virtual machine configurations than system states or a list of exploits, as these configurations are descriptions of the devices and their content. Furthermore, host-based model where attackers hop from one device or account to another are directly representative of the behavior of an actual attacker within a network. This is not to suggest that vulnerability or system-based approaches are invalid; rather, it is to suggest that they may be more appropriate for different applications, such as network vulnerability analysis, than for the concrete application of designing an attack scenario and the path of an attacker within it.

Relevant work includes Ammann *et al. ibid.*, whose model is intended for use by analysts who wish to identify attack paths within an architecture. In this work, nodes correspond to machines and transitions correspond to access levels between two hosts. For instance, assuming two machines named "Database" and "Web", the latter being vulnerable to a remote exploit providing root privileges and being accessible from "Database", a transition will be added between these two machines labeled with "root" and other information related to the exploit such as its name or the port used. A graphical representation of such a graph is available in figure 1.11.

Also intended for analyst use but with an emphasis on reducing graph complexity for large networks, Zhong *et al.* Shangqin Zhong, Danfeng Yan, and Chen Liu, « Automatic Generation of Host-Based Network Attack Graph », *in: 2009 WRI World Congress on Computer Science and Information Engineering*, vol. 1, 2009, pp. 93–98, DOI: 10.1109/CSIE.2009.102 have a different approach: nodes still correspond to machines, but transitions are no longer related to access levels. Instead, they are labeled with information about the attack, such as conditions (i.e. requirements on the attacker or the machine for the attack to be executed) or effects on destination, alongside a difficulty level.

One of the most influential work on this thesis was Mensah's model Mensah, *op. cit.*, designed for the modelization of cloud infrastructure attacks. Nodes hold information about not only the device the attacker is currently logged on, but also their level of privilege on this connection. This is in a way reminiscing of the formalism of Ammann *et al.* Ammann *et al.*, *op. cit.*, except privileges are now part of the nodes rather than the transitions. This makes it easier to monitor the position and abilities of an attacker at any given time and reduces the load of information held by transitions, at the cost of multiplying the number of nodes. A node may also indicate whether it contains interesting data for the attacker, such as a password or a secret file. An attacker may move in this model by taking a transition between 2 nodes, which may have preconditions or consequences on the state of the devices and the attacker. In order to model the fact that some transitions require that the attacker has accomplished specific actions in the network before (such as finding a password), some transitions have triggers attached to them. These triggers indicate that a transition can only be taken if another specific node has been visited by the attacker beforehand. An example can be seen in figure 1.12. Nodes are tuples of (Machine, User), except for H3 which also contains data for the attacker to acquired (keyUser1). Transitions are either CVEs to exploit or more complex actions, some of which require data to be acquired beforehand. Mensah's model also takes into account attack conditions and effects

on destinations - similar to the work of Zhong *et al.* Zhong, Yan, and Liu, *op. cit.* In the examples provided by Mensah, conditions appear to be related to attacker location (i.e. which node they currently control) and whether they accomplished actions related to triggers beforehand.

This approach offers several advantages in the context of cyber range description and deployment. First, the choice of using (User, Machine) couples for nodes is easy to translate into virtual machine instructions, which typically install software or add files to specific users. Second, the notion of triggers - transitions that necessitate the completion of prior steps before proceeding — may force the attacker to move around the architecture, potentially increasing the length and complexity of a given challenge. It is also realistic, as credential gathering before lateral movement onto other machines (to use said credentials) is a common step in real life attacks lockheedmartin, *op. cit.*

This model isn't however directly applicable for our purposes. This isn't a criticism of the model per se, but rather an acknowledgment that it was designed with a different goal than ours in mind. Storing data for the attacker in the node itself may also prove itself to be awkward. What if the information is indeed stored in a specific (User, Machine) position but requires an additional step (for instance the exploitation of a CVE) to be acquired? More importantly, the transition labeling lacks a clearly defined set of values. For instance, the example showcased in 1.12 features "CVE-2019-1918" alongside the more ambiguous "CONNECT". A defined ensemble of actions the attacker may take - or at least a more internally coherent one - would be valuable in the context of a scenario-designing tool.

The work of Berady *et al.* Aimad Berady et al., « PWNJUTSU: A Dataset and a Semantics-Driven Approach to Retrace Attack Campaigns », *in: IEEE Transactions on Network and Service Management*, Special Issue on Recent Advances in Network Security Management 19.4 (2022), pp. 5252–5264, DOI: 10.1109/TNSM.2022.3183476, URL: <https://inria.hal.science/hal-03694719> presents a few similarities to the work of Mensah Mensah, *op. cit.* but expands on them in relevant ways. Their goal is to model existing attacks and attack patterns in order to effectively label datasets and represent the path of an attacker throughout the experiment. In this work, a machine M is defined as:

- A set of services alongside their port, for instance *Apache(port: 443)*.
- A set of tuples (p, U) , with p a path to files and U a set of users who have access to these files.
- A set of tuples $(User, Service, Key, Level of Privilege)$ which specifies the required

credentials (here *Keys*) to access a service, alongside the privileges granted by said service.

- A set of other machines that can be reached from M.

In addition to this representation of the architecture, Berady et al. also propose a method for representing the current state of an attacker. They initially define the attacker's knowledge of a machine as a subset of the machine's description. That is to say, the attacker is aware of and has access to a subset of services, files, credentials, and adjacent machines. The state of an attacker is then represented as:

- A set of all pairs of machine and user (M,u) that the attacker has access to. Note that this is similar to the previously mentioned approach of Mensah's work *ibid*.
- A set of file contents containing secrets (i.e. password, keys...) relevant to the attacker's progression.
- The attacker's knowledge of each machine, as defined above.

Finally, the progression of an attacker throughout the network is represented by transitions labeled with techniques from the MITRE ATT&CK ENTERPRISE matrix MITRE, *op. cit*. The functioning, intricacies, perks and goals of this matrix will be detailed in a later section, but it is in short a nomenclature to describe attacker practices and techniques. Taking a transition - i.e. applying an attack technique - changes the state of the attacker, by providing them with new user sessions, secrets, or improving his knowledge of a machine. A visual representation of this model is available in figure 1.13, which showcases the three aspects of this representation: the attacker's abilities, their state, and the state of each machine.

In terms of the high/low levelness of their work, this model still requires the scenario designer to explicitly write out which specific service runs on each port or specify files to be added to machines. As such, their work is not directly applicable for the purposes of this thesis. Some of the concepts mentioned - such as the use of MITRE ATT&CK techniques to label transitions, the concept of secrets or the representation of an attacker's knowledge - will be mentioned again throughout this thesis. This may be caused by the fact their work was run at the same research team as the one of this author's thesis, which lead to some discussions about models and may have influenced our own model. However, there are significant differences between the two, resulting from the different scopes of applications. The former model focuses on the modeling of existing attack scenarios for analysis, whereas the latter model is designed for the modeling of attack scenarios to

deploy vulnerable architectures.

Finally, MITRE released in 2022 the Attack Flow model. Intended for easier communication and representation of existing attack scenarios for the benefit of defenders, Attack Flow represents scenarios as a graph of MITRE ATT&CK techniques², similarly to a vulnerability-based model. Techniques may also be connected to conditions, which represent plain text explanations of an attacker's required knowledge or data in order to progress throughout the scenario. Figure 1.14 showcases an example of such a condition. Here, the attacker may perform one of two techniques. They may dump the LSASS memory of the machine in order to harvest plaintext credentials, or they may create a local user account. Once this is performed, the attacker would gain the ability to add Autostart Execution to that account. This action would be possible because both of the two possible techniques beforehand provide the attacker with access to a local account, which is the actual requirement to perform the Autostart Execution attack. Thus, in order to make this clearer to the reader, a condition is added to the graph between the techniques. Finally, some actions may provide rewards to attackers once performed in the form of assets, such as credentials or access to specific services.

The Attack Flow model appears to be particularly pertinent for the purpose of organizing information regarding existing attack scenarios, such as those delineated in text attack reports. Moreover, as a MITRE framework, it boasts inherent compatibility with other MITRE tools. For instance, an Attack Flow representation can be directly mapped to the ATT&CK matrix, thereby offering a distinct perspective on the set of techniques employed. Additionally, it provides open-source visualization and graph building tools. However, it is important to note that while Attack Flow may offer certain advantages in the modeling of attack scenarios intended for deployment with it, it does not directly represent the current position of an attacker within a network or the structure of that network itself. As a result, while Attack Flow applications were considered throughout this thesis, for instance, to offer a more graphical representation of scenarios modeled using our own language, it ultimately was built for a different purpose than ours and is thus not directly applicable to our case.

We will now look into the question of the labeling of an attack step, i.e a single transition between two positions within the network and how the attacker goes from one to the other. Indeed, this was our main issue with using the work of Mensah Mensah,

2. Users may add their own techniques or use other frameworks, but MITRE existing techniques is the default and most commonly used mode.

op. cit. for our purposes. We will look into the available state of the art's answers to this question in the next section, with in particular the MITRE ATT&CK ENTERPRISE matrix MITRE, *op. cit.* used by Berady *et al.* Berady et al., *op. cit.*

1.2.2 Attack step representation.

When attempting to describe an attack, one may consider detailing attacker action on a low level: which vulnerabilities were used, the specific path taken within the network, etc. This approach, while valuable, makes it difficult to compare different attacks, as the description is tied to the specific architecture that was attacked. There is thus an incentive to develop higher level description tools for attacks, which would describe actions in a broader stroke without dwelling too much on the technical details. While higher-level representations may not inherently possess superiority, their generic nature does come with a trade-off in terms of precision. However, describing attack actions in such a way makes it easier to compare different attack scenarios or to classify attacks into larger categories. This may for instance make it possible to recognize an attacker's *modus operandi* between two different attacks if they follow similar patterns in their actions, despite the architectures being different. These kinds of comparisons do however require scenario descriptions to be precise enough, and ideally to be expressed in a similar description language. Indeed, while textual reports (such as the ones found in this compendium of APT reports *aptnotes*, *APTnotes data*, URL: <https://github.com/aptnotes/data> are very valuable from a research standpoint, they may not be directly parsable to a formalized description unless attack steps are expressed in a normalized language.

A higher level attack description is also particularly relevant for this thesis' goal, and having a fixed nomenclature for it may prove useful in solving our main gripe with the work of Mensah Mensah, *op. cit.* described above: the lack of a fixed possible set of attacker actions.

The Cyber Kill Chain lockheedmartin, *op. cit.* is one of the first influential works on the topic. Inspired by the military concept of a kill chain - which identifies the structure of an attack, such as identification or destruction of a target -, the Cyber Kill Chain identifies 7 main steps in an attack:

- Reconnaissance: Harvesting data about the target (email addresses, network structure...)
- Weaponization: Creating a remote access malware weapon, tailored to one or more

vulnerabilities.

- Delivery: Providing the weaponized bundle to the victim (phishing email, USB, fraudulent website...)
- Exploitation: Triggering the malware weapon in order to execute code on the victim's system.
- Installation: Installing additional malware and/or backdoor on the asset.
- Command and Control: Assuming remote control of the victim.
- Actions on Objectives: Accomplishing their original goal (encryption, data exfiltration...).

As this is the only attack framework provided by it, the Cyber Kill Chain suffers from a lack of flexibility. The "Weaponization" step in particular seems specific and not necessarily applicable to every attack, and this step alongside the Reconnaissance one are typically executed outside the target network. Furthermore, this model is strictly focused on malware and payloads (ignoring for instance web-based entry points), and lacks the ability to model insider threats such as APTs. These weaknesses made it unfit for this thesis, as it would limit the kind of scenario we would be able to deploy too harshly, while being ultimately a bit too vague for our purposes.

In 2013, the MITRE corporation proposed the MITRE ATT&CK® Matrix for Enterprise MITRE, *op. cit.*, a cyber-adversary behavior knowledge base aiming to describe the different steps of a given attack scenario. Although the MITRE has released multiple matrices, including one for mobile device attacks, this thesis will refer to the enterprise version as the "ATT&CK Matrix" for simplicity. This nomenclature can be regarded as an extension and improvement over the previously mentioned Cyber Kill Chain. It is able to describe attacker actions on three levels of abstraction, known as Tactics, Techniques and Procedures, or TTPs.

- Tactics represent the main goal of an action performed by an attacker³. For instance, an attacker may try to gain higher-level permissions on a system or network (TA004: *Privilege Escalation*).
- Techniques represent the means used by an attacker to reach his tactical goals. For instance, an attacker may exploit a vulnerability to elevate his privileges on a machine (T1068: *Exploitation for Privilege Escalation*⁴).

3. This is arguably the MITRE's own version of the Cyber Kill Chain.

4. We are here using the official MITRE numbering for techniques.

- Procedures describe the specific implementation details of an attack. For instance, an attacker may attempt to exploit a flaw in the `pkexec` process on older Linux versions NVD, *CVE-2021-4034 Detail*, 2022, URL: <https://nvd.nist.gov/vuln/detail/CVE-2021-4034> (visited on 08/30/2023) in order to escalate his privileges on a machine.

The ATT&CK matrix currently comprises 14 tactics, over 200 techniques and describes a few procedures for each technique. An overall view of the matrix, represented as a table of techniques grouped by tactics, can be seen in figure 1.15. Figure 1.16 showcases the page related to technique T1068: *Exploitation for Privilege Escalation*. This includes a description of the technique, examples of procedures alongside known attacker groups who used it, and information about the required permissions or platforms related to the technique. Note that this page is one of the more complete ones: the amount of information provided varies strongly between techniques. Some of the techniques are further divided into subtechniques: for instance T1552 (Unsecured Credentials) is divided into 8 different subtechniques, such as T1552.001: *Credential In Files* or T1552.003: *Bash History*, providing if needed a lower level of description. Subtechniques will however be mostly ignored in the work of this thesis for consistency reasons, as most techniques lack subtechniques.

There is to this author’s knowledge no exhaustive list of procedures for a given technique. The work of Hemberg et al Erik Hemberg et al., *Linking Threat Tactics, Techniques, and Patterns with Defensive Weaknesses, Vulnerabilities and Affected Platform Configurations for Cyber Hunting*, 2021, arXiv: 2010.00533 [cs.CR] is the closest we could find, as it does manages to connect every single nomenclature presented by MITRE (Tactics, Techniques, but also CVEs or Common Weakness Enumerations (CWE)) into a single graph. This makes it possible to occasionally have a list of CVEs for a given technique and thus provides a list of procedures (apply given CVEs) for these techniques. This is not however akin to an exhaustive list of procedures for our purposes, as it does not take into account procedures not linked to CVEs (of which there are plenty). Furthermore, the link between Techniques and CVEs is incomplete: plenty of techniques do not have any CVE associated to them and vice versa, due to limitations in the reliance on MITRE’s nomenclatures and the fact some techniques (for instance, T1518: *Software Discovery*) do not necessarily require any vulnerability to be accomplished.

Despite these limitations, we found the ATT&CK matrix to be particularly relevant for the work of this thesis for two major reasons:

- It is a well known and standardized format, used by several other projects in the literature. For instance, Kuppa *et al.* Aditya Kuppa, Lamine Aouad, and Nhien-An Le-Khac, « Linking CVE's to MITRE ATT&CK Techniques », *in: Proceedings of the 16th International Conference on Availability, Reliability and Security, ARES '21*, Vienna, Austria: Association for Computing Machinery, 2021, ISBN: 9781450390514, DOI: 10.1145/3465481.3465758, URL: <https://doi.org/10.1145/3465481.3465758> use neural networks to map Common Vulnerabilities and Exposures (CVE, a nomenclature for documenting known information-security issues) to MITRE ATT&CK techniques. Sharmet *et al.* Yashovardhan Sharma, Simon Birnbach, and Ivan Martinovic, *RADAR: A TTP-based Extensible, Explainable, and Effective System for Network Traffic Analysis and Malware Detection*, 2023, arXiv: 2212.03793 [cs.CR] also use machine learning, but use it to map network logs onto MITRE TTPs instead. Outkin *et al* Alexander V. Outkin et al., « Defender Policy Evaluation and Resource Allocation With MITRE ATT&CK Evaluations Data », *in: IEEE Transactions on Dependable and Secure Computing* 20.3 (2023), pp. 1909–1926, DOI: 10.1109/TDSC.2022.3165624 model attacks using Markov chains and MITRE techniques in order to evaluate attacker and defender optimal actions in a game theory perspective. It is also used in attack reports and emulation compendiums MITRE, *APT3 Adversary Emulation Plan*, 2018, URL: http://attack.mitre.org/docs/APT3_Adversary_Emulation_Plan.pdf (visited on 11/20/2023). While these projects ultimately have different scopes and goals than this thesis, reusing the ATT&CK matrix makes any potential future combination of these tools easier.
- The Technique level of the ATT&CK matrix is a reasonable compromise for our purposes of describing attacker actions. Indeed, such descriptions are specific enough to have an understanding of the attack - hence the use of this format in plenty of attack reports -, but overall remains platform and configuration agnostic. The tactic and procedural levels also have their uses (the latter in particular), which will be discussed in a later section.

Alongside a description of a few procedures, each technique may also provide the list of platforms susceptible to be attacked by this technique, attacker groups known to have used this technique in the past, or indications on how to detect its execution. The ATT&CK matrix is still being regularly updated, with the MITRE deploying new versions every few months. These changes include the addition, removal or reworking of existing techniques. For instance, MITRE ATT&CK v13 MITRE, *ATT&CK v13*

Enters the Room: Pseudocode, Swifter Search, and Mobile Data Sources, 2023, URL: <https://medium.com/mitre-attack/attack-v13-enters-the-room-5cef174c32ff> (visited on 11/20/2023) improved Linux coverage by adding several new techniques and subtechniques related to the platform.

1.2.3 Conclusion of State of the Art.

The topic of cyber range modelization and deployment is evidently on the rise these past years: most of the publications we covered have been made in the past 5 years. These projects are able to design and deploy vulnerable architectures based on a scenario description, but this description is often low-level, requiring the cyber range designer to specify exactly which vulnerabilities, operating systems or software have to be installed on each machine. Some works Venkatesan et al., *op. cit.*; Janisch, Pevný, and Lisý, *op. cit.* do offer ways to decline a higher-level description into low-level deployments, but their approaches are still tied to specifying services or exploits to be added on the machines and could thus be pushed even further. Thus, there is an incentive to have a new modelization pushing this concept even further, as this would allow scenario designers a greater flexibility in how they model and deploy architectures, as well as providing attackers with more variability. Our proposed solution for this thesis is a new scenario description formalism. Compared to the state of the art , it is to be able to describe a single scenario on a high-level in order to later refine that description into several different instances of that scenario. These instances will differ in their implementation but all follow the same path as the original scenario. This gives us the flexibility of the high-level representation, making it easier for scenario designers to write or tweak scenarios without having to directly modify configurations for each machine.

Existing attack graph models were considered for our modelization, as these offer a wide array of ways to represent an attack scenario depending on the use case. Most of these models however are not intended for the purpose of architecture deployment, but rather for the modelization of existing architectures or attacks, giving them a level of exhaustivity which would be ill-suited for our purposes. This includes the work of Mensah Mensah, *op. cit.*, a host-based representation, which was designed in order to represent cloud network attacks for defense analysis. Some of its basic choices of representations however, in particular in how it chose to represent attack positions, were relevant to our goal. There was however a need for a fixed set of attacker actions. The ATT&CK matrix appeared to be an industry standard on the topic. Indeed, it is widely used in the literature,

still under active development and linked to other MITRE nomenclatures (such as CVEs), making it particularly adapted to this thesis.

In the next section we will describe the design of the formalism used throughout this thesis in order to develop URSID, our own approach on a vulnerable architecture deployment tool. It combines attack graphs - in particular taking inspiration from Mensah's *ibid.* work - and the ATT&CK matrix MITRE, *Enterprise Matrix* in order to achieve a modular, high-level and easy to use attack scenario formalism that is adapted for the purpose of cyber range deployment. The main goal of this formalism compared to the state of the art is to be able to describe a single scenario on a high-level in order to later refine that description into several different instances of that scenario. These instances will differ in their implementation, but all follow the same path as the original scenario. This improves the replayability of the scenario, as all those instances offer a different challenge to the attacker without the need to rewrite the original scenario description. Meanwhile, the fact this description is done on a high-level make it easier to tweak or create new scenarios compared to the state of the art.

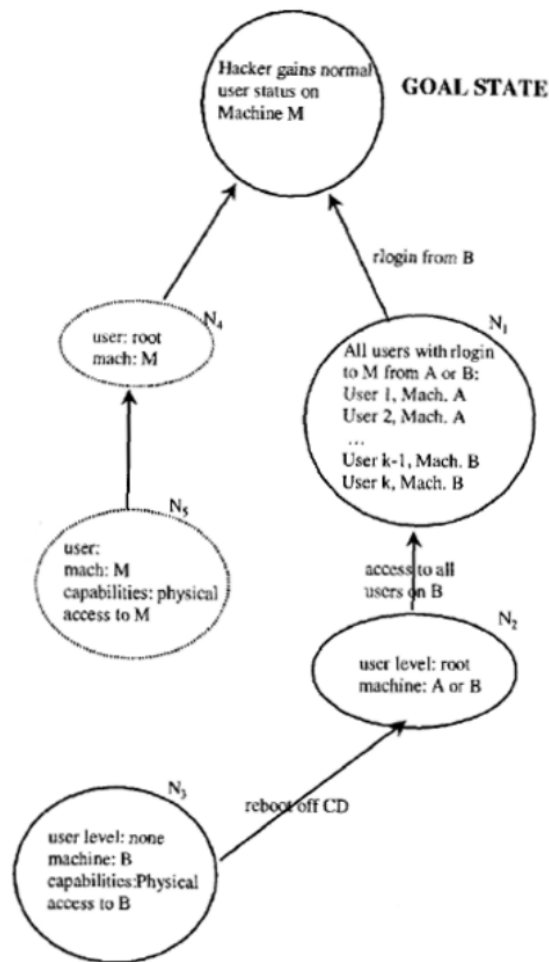


Figure 1.9 – Example of state based attack graph, according to the work of Cynthia Phillips and Laura Painton Swiler, « A graph-based system for network-vulnerability analysis », *in: Proceedings of the 1998 Workshop on New Security Paradigms*, NSPW '98, Charlottesville, Virginia, USA: Association for Computing Machinery, 1998, pp. 71–79, ISBN: 1581131682, DOI: 10.1145/310889.310919, URL: <https://doi.org/10.1145/310889.310919>.

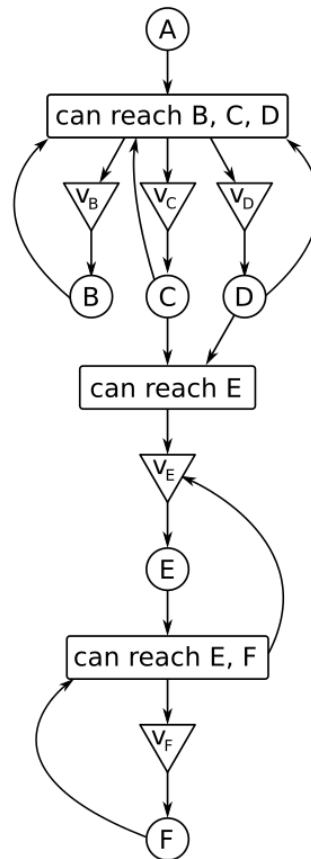


Figure 1.10 – Example of vulnerability based attack graph, according to the work of Kyle Ingols, Richard Lippmann, and Keith Piwowarski, « Practical Attack Graph Generation for Network Defense », in: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006, pp. 121–130, DOI: 10.1109/ACSAC.2006.39.

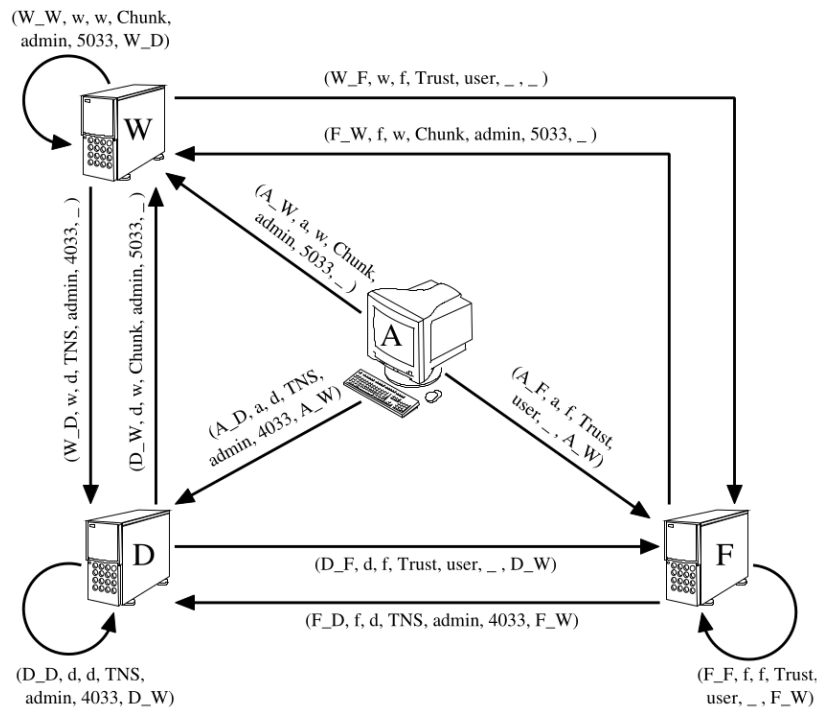


Figure 1.11 – Host-based attack graph according to the work of Ammann et al P. Ammann et al., « A host-based approach to network attack chaining analysis », in: *21st Annual Computer Security Applications Conference (ACSAC 05)*, 2005, 10 pp.–84, DOI: 10.1109/CSAC.2005.6.

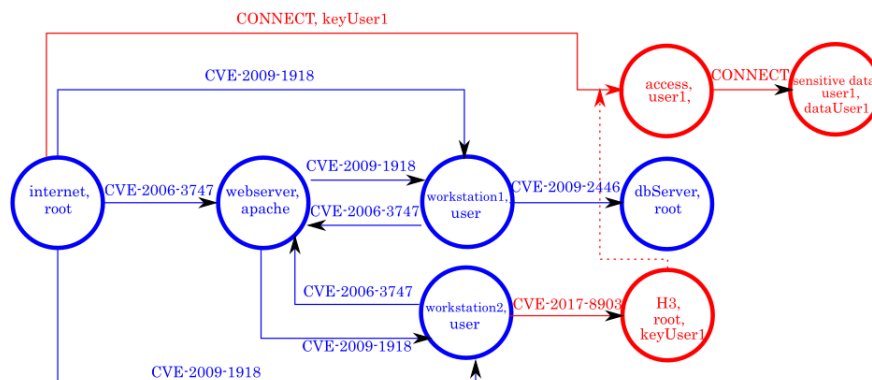


Figure 1.12 – Representation of an attack graph as seen in Mensah's Pernelle Mensah, « Generation and Dynamic Update of Attack Graphs in Cloud Providers Infrastructures », Theses, CentraleSupélec, June 2019, URL: <https://inria.hal.science/tel-02416305> work.

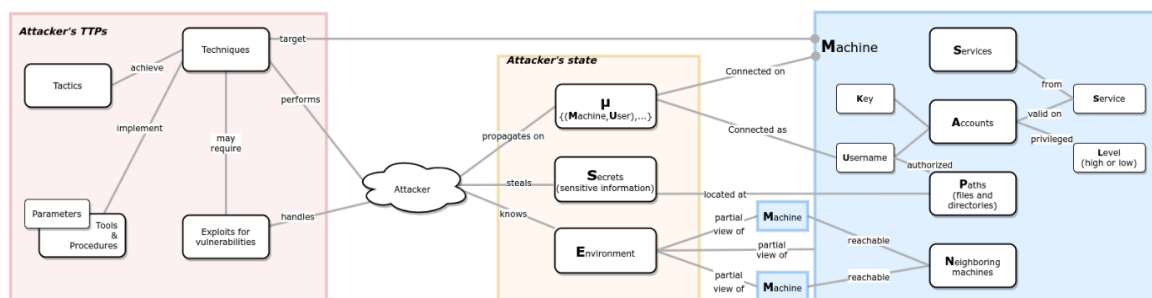


Figure 1.13 – Representation of Berady *et al* attack model, as seen in Aimad Berady et al., « PWNJUTSU: A Dataset and a Semantics-Driven Approach to Retrace Attack Campaigns », in: *IEEE Transactions on Network and Service Management*, Special Issue on Recent Advances in Network Security Management 19.4 (2022), pp. 5252–5264, DOI: 10.1109/TNSM.2022.3183476, URL: <https://inria.hal.science/hal-03694719>.

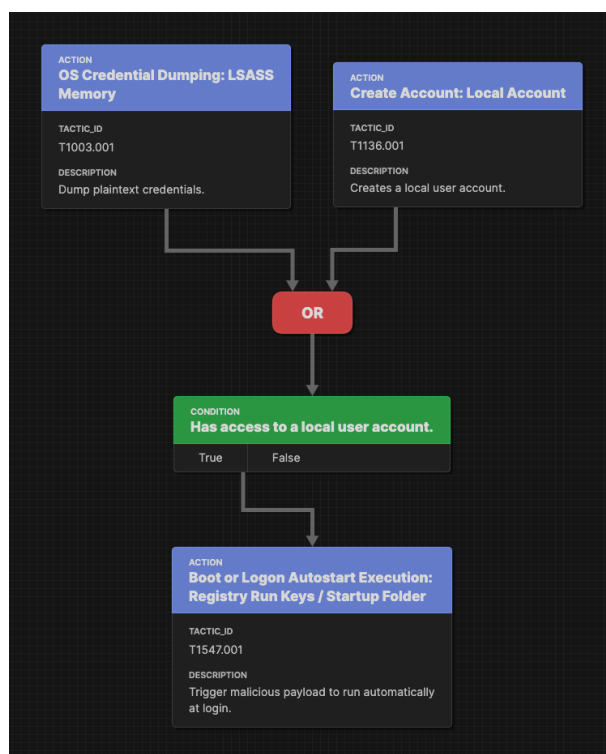


Figure 1.14 – Attack Flow representation of a sequence of techniques with a condition (taken from MITRE, *Attack Flow*, 2022, URL: <https://center-for-threat-informed-defense.github.io/attack-flow/overview/>).

Reconnaissance 10 techniques	Resource Development 8 techniques	Initial Access 10 techniques	Execution 14 techniques	Persistence 20 techniques	Privilege Escalation 14 techniques	Defense Evasion 43 techniques	Credential Access 17 techniques	Discovery 32 techniques
Active Scanning (3)	Acquire Access	Content Injection	Cloud Administration Command	Account Manipulation (6)	Abuse Elevation Control Mechanism (3)	Abuse Elevation Control Mechanism (3)	Adversary-in-the-Middle (3)	Account Discovery (4)
Gather Victim Host Information (4)	Acquire Infrastructure (6)	Drive-by Compromise	Command and Scripting Interpreter (9)	BITS Jobs	Access Token Manipulation (3)	Access Token Manipulation (3)	Brute Force (4)	Application Window Discovery
Gather Victim Identity Information (3)	Compromise Accounts (23)	Exploit Public-Facing Application	Container Administration Command	Boot or Logon Autostart Execution (14)	Account Manipulation (6)	BITS Jobs	Credentials from Password Stores (6)	Browser Information Discovery
Gather Victim Network Information (6)	Compromise Infrastructure (27)	External Remote Services	Deploy Container	Boot or Logon Initialization Scripts (3)	Boot or Logon Autostart Execution (14)	Debugger Evasion	Exploitation for Credential Access	Cloud Infrastructure Discovery
Gather Victim Org Information (4)	Develop Capabilities (4)	Hardware Additions	Exploitation for Client Execution	Browser Extensions	Boot or Logon Initialization Scripts (3)	Deobfuscate/Decode Files or Information	Forced Authentication	Cloud Service Dashboard
Phishing for Information (4)	Establish Accounts (23)	Obtain Capabilities (6)	Inter-Process Communication (2)	Compromise Client Software Binary	Create or Modify System Process (4)	Deploy Container	Forge Web Credentials (2)	Cloud Service Discovery
Search Closed Sources (2)	Stage Capabilities (3)	Replication Through Removable Media	Native API	Create Account (3)	Domain Policy Modification (2)	Direct Volume Access	Input Capture (4)	Cloud Storage Object Discovery
Search Open Technical Databases (3)	Supply Chain Compromise (3)	Scheduled Task/Job (3)	Scheduled Task/Job (3)	Create or Modify System Process (4)	Domain Policy Modification (2)	Execution Guardrails (1)	Modify Authentication Process (3)	Container and Resource Discovery
Search Open Websites/Domains (3)	Trusted Relationship	Serverless Execution	Shared Modules	Event Triggered Execution (13)	Escape to Host	Exploitation for Defense Evasion	Multi-Factor Authentication Request Generation	Debugger Evasion
Search Victim-Owned Websites	Valid Accounts (4)	Software Deployment Tools	System Services (2)	External Remote Services	Event Triggered Execution (16)	File and Directory Permissions Modification (2)	Multi-Factor Authentication Request Generation	Device Driver Discovery
				Hijack	Exploitation for Privilege Escalation	Hide Artifacts (11)		Domain Trust Discovery
						Hijack Execution		File and Directory Discovery
								Group Policy

Figure 1.15 – MITRE Enterprise ATT&CK matrix, showcasing tactics and techniques. Some procedures are provided as an example when clicking on a specific technique.

Exploitation for Privilege Escalation

Adversaries may exploit software vulnerabilities in an attempt to elevate privileges. Exploitation of a software vulnerability occurs when an adversary takes advantage of a programming error in a program, service, or within the operating system software or kernel itself to execute adversary-controlled code. Security constructs such as permission levels will often hinder access to information and use of certain techniques, so adversaries will likely need to perform privilege escalation to include use of software exploitation to circumvent those restrictions.

When initially gaining access to a system, an adversary may be operating within a lower privileged process which will prevent them from accessing certain resources on the system. Vulnerabilities may exist, usually in operating system components and software commonly running at higher permissions, that can be exploited to gain higher levels of access on the system. This could enable someone to move from unprivileged or user level permissions to SYSTEM or root permissions depending on the component that is vulnerable. This could also enable an adversary to move from a virtualized environment, such as within a virtual machine or container, onto the underlying host. This may be a necessary step for an adversary compromising an endpoint system that has been properly configured and limits other privilege escalation methods.

Adversaries may bring a signed vulnerable driver onto a compromised machine so that they can exploit the vulnerability to execute code in kernel mode. This process is sometimes referred to as Bring Your Own Vulnerable Driver (BYOVD).^{[1][2]} Adversaries may include the vulnerable driver with files delivered during Initial Access or download it to a compromised system via [Ingress Tool Transfer](#) or [Lateral Tool Transfer](#).

ID: T1068

Sub-techniques: No sub-techniques

Tactic: Privilege Escalation

Platforms: Containers, Linux, Windows, macOS

Permissions Required: User

Effective Permissions: User

Contributors: David Tayouri, Idan Revivo, @idanr86, Team Nautilus Aqua Security, Joas Antonio dos Santos, @C0d3Cr4zy, Inmetrics, Yaniv Agman, @AgmanYaniv, Team Nautilus Aqua Security

Version: 1.5

Created: 31 May 2017

Last Modified: 07 April 2023

[Version Permalink](#)

Procedure Examples

ID	Name	Description
G0007	APT28	APT28 has exploited CVE-2014-4076, CVE-2015-2387, CVE-2015-1701, CVE-2017-0263 to escalate privileges. ^{[3][4][5]}
G0016	APT29	APT29 has exploited CVE-2021-36934 to escalate privileges on a compromised host. ^[6]

Figure 1.16 – Example of a technique as showcased by the ATT&CK matrix, here for T1068: Exploitation for Privilege Escalation.

SCENARIO DESCRIPTION FORMALISM.

In this chapter, we will present our choice of formalism for the description of vulnerable architectures. This formalism is inherently linked to the description of an attack scenario. In the context of this work, we define an attack scenario as a description of an architecture, which is vulnerable to a sequence of attacks referred to as attack steps. This description thus encompasses the structure of the network itself (i.e which machines populate it and how they are connected) alongside the vulnerabilities found within and between these machines. One possible analogy would be that of a traditional role-playing game: a given challenge (attack scenario) consists of both the design of the dungeon (architecture) and the dangers faced by the player in order to move around the dungeon (attack steps). We expand on the notion of attack graphs we explored in the state of the art in order to formally describe scenarios. This new formalism attempts to solve the issues we identified in the state of the art by providing a high-level description of attack scenarios, while still being flexible enough to be converted later in low-level description for deployment purposes. This formalism was implemented as a basis for our vulnerable architecture deployment platform, URSID. A summary of the formalism detailed in this chapter was published and presented at FPS 2023 Pierre-Victor et al., *op. cit.*

2.1 Goals of this formalism.

Our goal for this design was to provide a new way of solving the state of the art's perceived weakness in terms of re-usability of a given scenario description caused by their low-level nature. In order to fulfill this objective, perceived design specifications were as follows:

1. Scenario descriptions should be as concise as possible in order to minimize required work from the scenario designers.
2. Scenario descriptions should be generic enough to be platform and exploit agnostic in order to increase re-usability of a given description.

3. Scenario descriptions should be precise enough to be later converted into virtual machine and network descriptions.
4. The possible architectures resulting from this conversion process should offer some degree of variability while still corresponding to the same scenario description.
5. Attacker path and state within a scenario should be quantifiable in order to be tracked during the deployment step.

Points 2 and 3 are what lead to the use of the ATT&CK matrix MITRE, *op. cit.* Tactics, Techniques and Procedures. As presented in section 1.2.2 of the state of the art, use of the technical level of this nomenclature in order to label attack steps offers a good compromise between genericity and precision, all the while being an industry standard for attack reports. Point 3 implies a mechanism of conversion from a generic scenario description to one specific enough to be deployed by a virtual machine generation tool such as Vagrant Hashicorp, *Vagrant*. We thus introduce the notion of **refinement** of a scenario, which will deal with this conversion.

The objective of this refinement is to describe a scenario by labeling attack steps with MITRE ATT&CK techniques - i.e describing this scenario on a technical level - and then convert this description into one where attack steps are labeled with MITRE ATT&CK procedures - i.e describing this scenario on a procedural level. Since one technique may correspond to multiple procedures, this means that one technical level scenario may correspond to multiple procedural level scenarios, thereby satisfying point 4.

Design approaches based on describing attack steps on a tactical or procedural level were attempted, but ultimately discarded for being too generic or too specific, respectively. Describing a scenario on a procedural level would run into the same issues at the state of the art in terms of re-usability of a given description. Meanwhile, while a tactical level approach might theoretically work, we felt that this would lead to too much variation within a single description. For instance, let us consider two techniques associated to tactic TA0004: *Privilege Escalation*:

- T1078: *Valid Accounts*, in which an attacker manages to gain access to an existing privileged account on a machine.
- T1611: *Escape to Host*, in which an attacker breaks out of a container in order to access the underlying host.

T1078 is applicable to pretty much any system, whereas T1611 implies that the attacker landed in a container application, a much more specific use case. This is not to say T1611

is not a valuable technique to study, but rather that describing an attack step in the scenario as being part of a Privilege Escalation tactic would lead to potentially wildly different architectures in the end. Furthermore, one desirable use case for URSID would be to deploy an architecture vulnerable to a known attacker *modus operandi*, which are usually described on a technical level in security reports.

Point 1 warrants its own discussion. Work such as Mensah’s cloud network modelization Mensah, *op. cit.* - particularly influential for this thesis - were created in order to describe existing architecture, to calculate potential weaknesses within the network and offer remediation. Such an approach requires an exhaustive description of the network: any part of the architecture not present within the model is a potential attack surface which will be missing from the final defense report. Thus, works that convert an existing network into a description have to make conscious choices on which part they choose to omit in order to not render their conclusions invalid. On the contrary, URSID aims to convert a description into an architecture. Writing every single parameter of the scenario would be a considerable investment for the designer, especially when most of it would not be relevant for the attacker to perform. As thus, our goal with URSID’s scenario formalism is to limit ourselves to describing the minimum amount of information necessary to play the scenario. This incentive in limiting the scope of the description - compared to works which require this exhaustivity in order to be effective - is one of the reasons we did not reuse pre-existing attack graph and scenario description tools, which were intended for different purposes.

2.2 Detailed formalism

In this section, we will showcase the formalism used by URSID in two steps. First, we will address the question of the representation of the attack scenario itself, using an attack graph representation. We will then show how the path of an attacker within this attack scenario can be formalized and discuss the winnability of a given scenario.

2.2.1 Scenario description

Designing a graph representation requires first to formally define its vertices and edges. We went for a host-based representation, in which vertices correspond to positions within the network. An **attack position** corresponds to a session of an attacker, under the

identity of a user on a machine of the compromised network. This choice of representation, also present in the work of Mensah *ibid.*, has several perks to it:

- It is accurate to the movement of an attacker within a network: they may gain access to machines by compromising accounts or creating their own (usually privileged) login sessions. Techniques belonging to tactics such as TA0001: *Initial Access* (getting a session within the network), TA0004: *Privilege Escalation* (getting a session with more privilege than the current one) or TA0008: *Lateral movement* (getting a session on another machine within the network) mesh particularly well with this representation.
- Virtual machine management and deployment tools such as Vagrant Hashicorp, *op. cit.* first create the machine itself then populate it with accounts (alongside other information such as files or software) according to instructions given by tools like Ansible Inc, *op. cit.* Getting this list of machines and accounts to create is trivial with this representation.

We first establish the set of **machines** and **users** within a scenario. Note that these do not need to be exhaustive - i.e represent every single user that will be available in the resulting architecture - for reasons we mentioned earlier.

Definition 2.2.1 (Machines and users). $\mathcal{M}$ is the set of machine names which will be considered for the representation of this scenario. $\mathcal{U}$ is the set of usernames corresponding to accounts on those machines which are needed to represent the scenario.

Vertices within our representation will thus correspond to attack positions, which we showcase in definition 2.2.2.

Definition 2.2.2 (Attack position). An **attack position** is a pair (m, u) where $m \in \mathcal{M}$ is a machine of the compromised network and $u \in \mathcal{U}$ a declared user on this machine.

For instance, a given machine named *Bear* may have two accounts accessible as part of the scenario: a user account *Alice* and a privileged account *Superuser*. These attack positions would be defined as $p_0 = (Bear, Alice)$ and $p_1 = (Bear, Superuser)$. Figure 2.1 offers a graphical representation of those two attack positions.

Among these attack positions, a **starting position** is a position controlled by default by the attacker at the beginning of the scenario and from which he will be able to play the scenario. The starting position is usually $(Attacker, Superuser)$ - we consider that the attacker has full control over their own machine(s) -, but could also be a machine from the network, in the case of an insider threat. A **winning position** is a position that is of

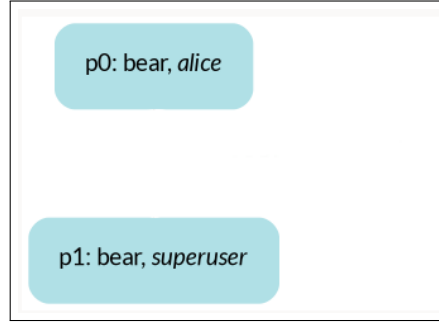


Figure 2.1 – Graphical representation of two attack positions.

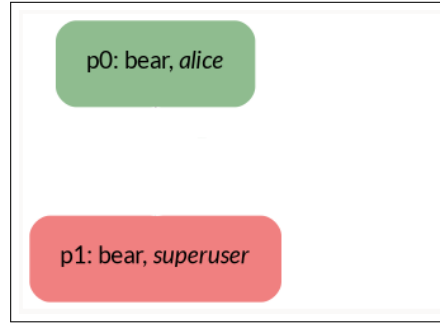


Figure 2.2 – Graphical representation of a starting and a winning attack position.

particular interest to the attacker and which corresponds to one of his ultimate goals in the architecture. For instance this may correspond to an account hosting sensitive data or the admin account of a Windows Active Directory domain controller. Note that there may be more than one starting or winning position within a scenario.

Definition 2.2.3 (Starting and winning positions). A **starting position** $\mathbf{S}$ is an attack position (m, u) representing a starting point of the attacker. A **winning position** $\mathbf{W}$ is an attack position (m, u) representing a goal of the attacker. Given a set of attack positions $\mathbf{P}$, $\mathbf{P}_s \subseteq \mathbf{P}$ and $\mathbf{P}_w \subseteq \mathbf{P}$ represent the set of starting and winning positions, respectively.

For instance, reusing the previous examples of $p_0 = (Bear, Alice)$ and $p_1 = (Bear, Superuser)$, we may define p_0 as a starting position and p_1 as a winning position. Figure 2.2 offers a graphical representation of those.

Now that we have represented positions within the network, we need a way to represent how the attacker moves within the network. In keeping with our choice of a host-based attack graph model, this movement will be formalized by edges between the nodes (attack

positions) of the attack graph. The progression of an attacker in a compromised network is made possible by the use of an attack technique, itself executed through an attack procedure according to the terminology proposed by the MITRE. This will be acknowledged in the formalism of the transitions by labeling them with either a technique or a procedure. This label will be referred to as τ throughout this thesis.

A given scenario can be formalized on a **technical or a procedural level** depending on the level of abstraction chosen for the labeling of its transitions. The technical level is a more high-level representation which attempts to answer the issues of the state of the art related to re-usability of a given representation. This procedural level is a more low-level representation containing enough specifics about the architecture to be converted in a virtual machine and deployed. Note that since a single MITRE technique is associated to several procedures, this implies that a single scenario represented on a technical level can be associated to multiple procedural representations, helping with scenario variability. We discuss the process of conversion (which we call refinement) of a technical representation into a procedural one in section 3.

An **attack scenario** thus describes a set of attack positions that can be reached by an attacker and how that attacker may progress between those attack positions. It is to be noted that our goal is not to monitor every single vulnerability on an existing architecture. Instead, it is to focus on the generation of an architecture vulnerable to a specific sequence of attacks, for instance one matching the workflow of a known APT.

A **transition** allows the attacker to move from two attack positions p_0 and p_1 in the network, by performing the technique or procedure τ depending on the level of abstraction. Note that p_0 and p_1 might be the same: some transitions may not lead directly to new attack positions, but instead provide them with additional data required to make progress through the scenario. This is what Mensah’s work Mensah, *op. cit.* attempted to solve via the use of triggers. In their work, some transitions within the network could only be taken if a specific precondition had been previously met, in the form of a trigger tied to another position within the network. We offer a different solution to this need of representation. First, instead of tying these accomplishments to an attacker landing in a position in the network (i.e. landing on a node) in the graph representation, we found it more appropriate to have these accomplishments be rewarded as a result of the transition itself. This change was made in order to account for transitions that take place on a single node.

As an example, let’s take a hypothetical attack position $p_0 = (root, machine)$, a privilege attack position which contains credentials for the attacker to recover. For instance, the

attacker may be expected to perform technique T1552: *Unsecured Credentials*, which encompasses way in which an attacker will find and obtain insecurely stored credentials. Performing this technique does not directly provide the attacker with a new attack position. However, there is still work required from the attacker in order to extract these credentials once he lands on p_0 . Thus, tying the acquisition of these credentials to the attack position itself would lead to a false assumption that the attacker is always in possession of these credentials the moment he got in that position, which is false. Tying these rewards to the execution of the technique instead solves this problem.

Some transitions within the network may thus reward the attacker with data, which they will then need in order to access further transitions. To reuse the game analogy above, this is akin to some challenges encountered by a player providing them keys needed to open doors further along the dungeon.

This brings us to definition 2.2.4:

Definition 2.2.4 (Edges). For a given scenario, the set of edges $\mathbf{A}^{\text{tech}}$ is composed of edges $(p_1, p_2, (\tau, \mathbf{pre}, \mathbf{post}))$ where:

- $p_1, p_2 \in \mathbf{P}$ are attack positions,
- τ is an attack technique (or an attack procedure depending on the level of abstraction),
- $\mathbf{pre} \in \mathcal{D}$ are the data required to execute τ
- $\mathbf{post} \in \mathcal{D}$ are the data granting the attacker when he successfully completes τ

Note that in most cases, this data correspond to passwords, keys or credentials in general. We will thus refer to them as **secrets** throughout this thesis.

For example, reusing the previously defined $p_0 = (\text{Bear}, \text{Alice})$ and $p_1 = (\text{Bear}, \text{Superuser})$, we may consider that the attacker has to first acquire a secret *secret* on p_0 using technique T1552: *Unsecured Credentials*. This secret - for instance a password stored in a text file - could then be used in order to access p_1 using T1078: *Valid Accounts*.

These transitions would be represented as $t_0 = (p_0, p_0, (T1552, [], [\text{secret}]])$ and $t_1 = (p_0, p_1, (T1078, [\text{secret}], []))$, respectively. A visual example of these is available in Figure 2.3.

Having defined nodes and edges, we may now define a scenario as definition 2.2.5.

Definition 2.2.5 (Attack scenario). An attack scenario over an architecture with $\mathcal{M}$ machines and $\mathcal{U}$ declared users is a graph $\mathbf{S}^{\text{tech}} = (\mathbf{P}, \mathbf{A}^{\text{tech}})$ where

- the set of nodes $\mathbf{P} \subseteq \mathcal{M} \times \mathcal{U}$ is a set of attack positions.

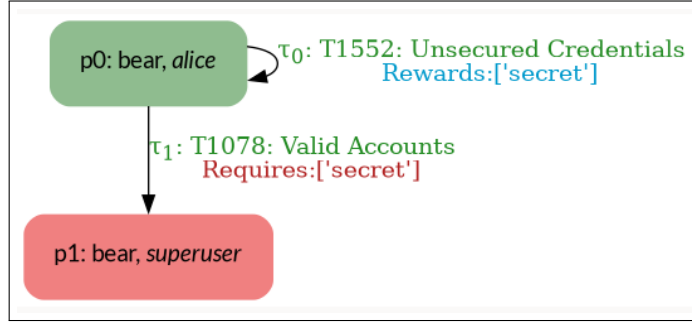


Figure 2.3 – Graphical representation of two transitions between two attack positions.

— the set of edges $\mathbf{A}^{\text{tech}}$ is $(p_1, p_2, (\tau, \mathbf{pre}, \mathbf{post}))$.

Since each edge in the set of edges $\mathbf{A}^{\text{tech}}$ can be defined on a technical or on a procedural level - depending on whether τ is a MITRE technique or procedure -, we can naturally define attack scenarios as being either on a technical or on a procedural level. Most of the time, however, the scenarios will be described on a technical level in order to provide the flexibility and variability in the deployed architectures that's at the core of URSID's mission statement. We will discuss in chapter 3 how to get from a technical to a procedural level description.

Now that we have defined scenarios as a set of nodes - representing attack positions within the networks - and transitions - representing techniques or procedures the attacker has to employ to move within the scenario or acquire knowledge -, we will now turn to the issue of describing the path of an attack within the scenario.

2.2.2 Attacker progression

Summarily, we choose to represent an attacker as the set of attack positions they have compromised and data they acquired. The attacker starts by controlling the starting positions $\mathbf{S}$. This could be either be the attacker's own machine in case of a remote attack, or a compromised workstation the attacker has physical access to in case of an offline attack. The attacker then walks along the graph edges and acquires attack positions by performing techniques, until they acquire a winning position. At each step, the attacker is in control of a set of attack positions, and acquired a set of secrets by taking transitions within the scenario. We refer to these sets as the attacker's **attack state**, which leads us to definition 2.2.6:

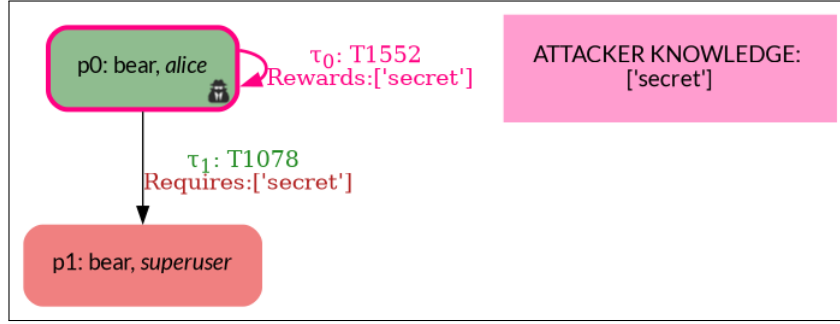


Figure 2.4 – Graphical representation of an attack state.

Definition 2.2.6 (Attack state). For a given scenario $\mathbf{S}^{\text{tech}} = (\mathbf{P}, \mathbf{A}^{\text{tech}})$, we model the attacker’s progression as **an attack state** defined by $\mathcal{A} = (\mathcal{P}, \mathcal{K})$, where:

- $\mathcal{P} \subseteq \mathbf{P}$ is the set of all attack positions they control.
- $\mathcal{K} \subseteq \mathcal{D}$ the data they acquired.

For example, in the previous scenario, an attacker may have taken control of the starting attack position $p_0 = (\text{Bear}, \text{Alice})$ before acquiring the data "secret" by performing the transition τ_0 . This would lead to an attack state $\mathcal{A} = ([p_0], ["secret"])$. A graphical representation of this attack state is available in figure 2.4

In our model, we assume that an attacker may not lose the control of an attack position after he acquires it once. Indeed, we suppose that once they manage to open a session on a given machine by taking a sequence of transitions, the attacker is able to get back to that attack position by applying the same sequence, or by using a technique related to the Persistence tactic. While this hypothesis may exclude some destructive attacks or unstable exploits, these represent a small fraction of attacks in practice and considering node acquisition to be permanent will make it easier to traverse the graph in later algorithmic treatments.

The attacker progresses in the scenario by controlling new attack positions or increasing his knowledge on the architecture. If an attacker controls an attack position p_1 and if in the attack scenario there exists a position p_2 , a technique τ , two data sets **pre** and **post** such that the attacker already knows **pre** or can progress from p_1 to p_2 by applying τ . If the execution of τ succeeds, the attacker is rewarded with **post**.

An attack state thus represents the state of an attacker within the network at a given moment, which they are likely to enhance by performing attacks in order to acquire new attack positions and secrets. Formally, this leads us to definition 2.2.7:

Definition 2.2.7 (Attack state progression). For a scenario $\mathbf{S}^{\text{tech}} = (\mathbf{P}, \mathbf{A}^{\text{tech}})$, given attack states $\mathcal{A}_i = (\mathcal{P}_i, \mathcal{K}_i)$ and $\mathcal{A}_{i+1} = (\mathcal{P}_{i+1}, \mathcal{K}_{i+1})$, as well as attack positions $p_i \in \mathbf{P}$ and $p_{i+1} \in \mathbf{P}$ with $p_i \in \mathcal{P}_i$ and $\mathcal{P}_{i+1} = \mathcal{P}_i \cup \{p_{i+1}\}$, an attacker can progress from $\mathcal{A}_i$ to $\mathcal{A}_{i+1}$ by applying a technique τ in scenario $\mathbf{S}^{\text{tech}}$ if:

- The attacker controls the attack position $p_i \in \mathcal{P}_i$.
- There is a transition to move from p_i to the position p_{i+1} i.e. $(p_i, p_{i+1}, (\tau, \mathbf{pre}, \mathbf{post})) \in \mathbf{A}^{\text{tech}}$.
- The attacker masters the prerequisite to apply the transition $\mathbf{pre} \in \mathcal{K}_i$.

In that event, the attacker will be rewarded the secret $\mathbf{post}$, giving us $\mathcal{K}_{i+1} = \mathcal{K}_i \cup \mathbf{post}$.

For example, starting from the attack state $\mathcal{A}_0 = ([p_0], [])$, the attacker can reach the figure 2.4 attack state $\mathcal{A}_1 = ([p_0], ["secret"])$ by taking transition τ_0 , which rewards him with a secret.

An attack through a scenario can thus be represented as a finite sequence of attack states. The attacker starts with the control of the starting attack positions and no knowledge, and each new transition they perform provides them with either new positions or new secrets. We refer to such a sequence of attack states as an **attack path**.

Definition 2.2.8 (Attack path). An attack path is a finite sequence of attack states $\mathcal{A}_0 \xrightarrow{\tau_0} \mathcal{A}_1 \xrightarrow{\tau_1} \dots \xrightarrow{\tau_n} \mathcal{A}_n$.

For instance, reusing the previously introduced examples, $\mathcal{A}_0 = ([p_0], [])$ and $\mathcal{A}_1 = ([p_0], ["secret"])$, $\mathcal{A}_0 \xrightarrow{\tau_0} \mathcal{A}_1$ is an attack path.

Obviously, not every attack path is valid: there has to be actual transitions between two attack states for each part of the sequence that lead to the desired state. We qualify valid attack paths - i.e. that correspond to a valid sequence of attack states - as **executable attack paths** according to definition 2.2.9.

Definition 2.2.9 (Executable attack path). An attack path $\mathcal{A}_0 \xrightarrow{\tau_1} \mathcal{A}_1 \xrightarrow{\tau_2} \dots \xrightarrow{\tau_n} \mathcal{A}_n$ is said to be executable if the attacker can progress from $\mathcal{A}_0$ to $\mathcal{A}_n$.

For example, $\mathcal{A}_0 \xrightarrow{\tau_0} \mathcal{A}_1$ is executable, as it is possible to transition from one to the other by taking τ_0 .

However, if we define $\mathcal{A}_2 = ([p_0, p_1], [])$, then $\mathcal{A}_0 \xrightarrow{\tau_0} \mathcal{A}_2$ is not executable, as taking τ_0 does not reward the attack with position p_1 . In fact, there is no way to reach p_1 without

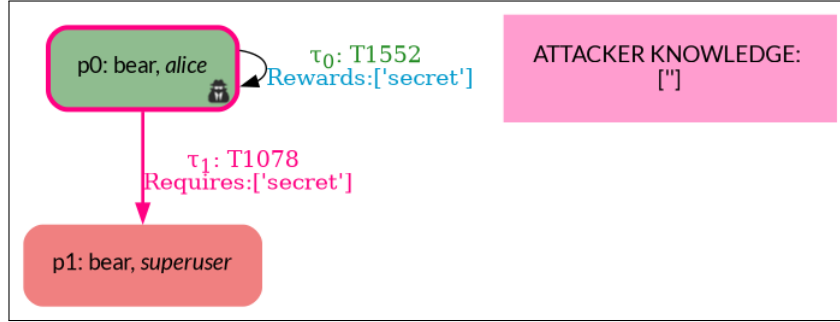


Figure 2.5 – Graphical representation of an attack state part of a non executable attack path. In fact, this state is impossible to reach within this scenario.

acquiring "secret", making this attack state impossible to reach. A visual representation of this attack state is available in figure 2.5.

The goal of the attacker is to acquire control of one of the winning attack positions $\mathbf{W}_e$, an endeavor which they begin with the control of one of the starting attack positions $\mathbf{S}_e$. Given these facts, in order for the scenario to be winnable, there needs to be at least one sequence of valid transitions for the attacker to take in order to reach one of the winning nodes. This brings us to definition 2.2.10

Definition 2.2.10 (Winnable scenario). A winnable attack scenario is a scenario presenting at least an executable attack path $\mathcal{A}_0 = (\mathcal{P}_0, \mathcal{K}_0) \xrightarrow{\tau_1} \mathcal{A}_1 \xrightarrow{\tau_2} \dots \xrightarrow{\tau_n} \mathcal{A}_n = (\mathcal{P}_n, \mathcal{K}_n)$, a starting attack position p_0 such that $p_0 \in \mathcal{P}_0$ and a winning position p_w such that $p_w \in \mathcal{P}_n$.

While it is easily possible to design an unwinnable scenario, one may question the point of doing so. Trivially unwinnable scenarios include those where there are no winning attack positions, where the winning positions have no transitions leading to them, or where the starting positions are isolated from the rest of the network. Secrets are also an important factor in the legitimacy of a given scenario: transitions requiring secrets become de facto impossible if no transition in the scenario rewards that secret.

Fortunately, it is easy (from a computational point of view) to figure out whether a given scenario is winnable or not. This is due to the fact that attack states can only be improved as the attacker progresses through the scenario, as we deliberately considered that attack positions cannot be lost once acquired. We consider the same fact to be true for secrets, because while there may be cases where secrets are temporarily valid or usable only once, we considered these cases to be too niche to consider, and preferred the simplicity of the model over complete exhaustiveness of the representation.

Furthermore, our representation of attack states makes it easy to determine whether an attacker is making progress in the network, since taking a transition should provide him with either new attack positions or secrets. If a transition provides neither - for example, if the attacker takes a transition a second time - it can be considered useless for the attacker's progress and discarded in the search for winning positions.

The outline of the winnability determination algorithm is as follows:

- Start with a beginning attack state in which the attacker holds starting attack positions and no secrets.
- Make a list of all transitions $(p_1, p_2, (\tau, \mathbf{pre}, \mathbf{post}))$ available for the attacker at the moment - i.e. the attacker controls p_1 and has the required secrets for it **pre** as part of their current attack state $\mathcal{A} = (\mathcal{P}, \mathcal{K})$.
- Remove all transitions that do not improve the attack state from consideration. This encompasses transitions in which the attacker already controls the ending node $p_2 \in \mathcal{P}$ and already knows all the secrets $Rewards \subseteq \mathcal{K}$
- Take a transition at random and repeat this process.
- Stop either when a winning attack position is reached, or there is no more transitions to take that advance the attacker state.

The complexity of this algorithm is trivially bounded by the number of transitions within the scenario, since in this model transitions already taken are never considered for a second round. Furthermore, no transition can regress an attacker's state, since we consider that both attack positions and secrets cannot be lost once acquired. This makes it fast enough for our applications, since transitions are written manually by the scenario designer anyway, and thus are unlikely to reach astronomically high numbers.

2.2.3 Conclusion on formalism

This section deals with URSID's attack scenario formalism. Using a host-based attack graph structure, scenarios are represented as a set of attack positions for the attacker to take control of, along with secrets they may need to acquire in order to progress. Meanwhile, a representation of the attacker, in terms of attack states and attack paths, allows us to evaluate his progress at any given time, in addition to ensuring that the scenario is winnable in the first place. This approach is not intended to be exhaustive, but rather to provide an efficient way for scenario designers to write out their plans. The proposed formalism provides two levels of granularity, either technical or procedural, according to

the MITRE TTP nomenclature. The technical level provides a high-level description of an attack scenario, allowing designers to plan their architecture without having to worry about specific implementation details. The procedural level, at a lower level, provides the details needed to deploy virtual machines. The goal of this distinction is to provide an answer to the perceived state of the art problems of scenario reusability and flexibility.

We will now offer a practical example of URSID's formalism in action, in order to showcase its abilities in representing attacks, inspired by the modus operandi of an actual threat actor.

2.3 Example: formalizing an APT3 inspired attack scenario.

In order to illustrate this formalism, we designed an example scenario to model, with the goal of being able to later deploy an architecture vulnerable to that attack scenario.

There were a few constraints in mind during the design:

- The scenario should have a few machines, enough to be nontrivial to model and interesting for the attacker to go through, but small enough to be developed, deploy and used as an example.
- The scenario should make use of secrets which force the attacker to visit some positions before others.
- The scenario should have more than one winning path, in order to showcase some of the abilities of this formalism.
- The techniques available in the scenario should be reasonably easy to implement later. This includes avoiding techniques that are exclusive to Windows operating systems (such as T1047: *Windows Management Instrumentation*), as Windows Virtual Machines are intrinsically harder to deploy and manage than Linux (configuration more based on a visual interface, licensing issues, automatic update issues, etc.).

We also wished for this scenario to be inspired by real life attacks, such as ones performed by Advanced Persistent Threat (APT) actors. Fortunately, and in part thanks to our choice of using the ATT&CK matrix, there is plethora of resources and attack reports out there detailing such attacks using the TTP framework.

One that particularly caught our attention was the MITRE APT3 Adversary Emulation scenario MITRE, *APT3 Adversary Emulation Plan*. According to MITRE, APT3 is a

China-based threat group, known to have targeted organizations in the US and Hong-Kong. This detailed report covers the tactics, techniques and procedures which were used by this threat actor in the wild, alongside information on how to reproduce these attacks locally by defenders for security training purposes.

This report identifies three phases in a typical APT3 attack campaign:

- **Initial Compromise.** The goal of this phase is for the attacker to achieve remote code execution and control of a system within the target network, in order to proceed with the rest of the attack. Known entry points used in real life attacks by APT3 include 0-day exploits in Internet Explorer, Windows or Flash, spear phishing campaigns and compromise of legitimate but weakly secured websites. For instance in 2014, APT3 actors sent spear phishing emails containing links to a malicious website, which then used an Internet Explorer CVE to launch and install payloads on the victim's machine and ultimately assume remote control.
- **Network Propagation.** The goal of this phase is for the attacker to spread throughout the network in order to take control of more devices and identify files or credentials of interest within the network. Tactics performed here include Discovery (with techniques such as T1083: *Discover Local System*), Privilege Escalation (T1068: *Exploitation for Privilege Escalation*, or Credential Access (T1552: *Unsecured Credentials*¹). The MITRE also provides actual code to replicate these attacks. Note that the report itself specifies that there is a lot of unknown around the actual methods used by APT3 for this phase, and that the researchers drew conclusions based on what limited data they had available.
- **Exfiltration.** The goal of this phase is for the attacker to extract valuable documents from the target. For instance, APT3 has been reported to compress documents using WinRAR before acquiring them through an encrypted connection on port 443.

Figure 2.6 offers a graphical representation of these 3 phases and the overall emulation plan, including all the techniques presented by the report.

As mentioned earlier, we had to make some concessions and tweaks when designing an attack scenario based on this emulation plan:

- Some of the techniques mentioned were not relevant to our goal of vulnerable architecture design. For instance, techniques related to the Exfiltration part of the

1. Some of these technique names or numbers have been updated from the report to match the current state of the ATT&CK matrix.

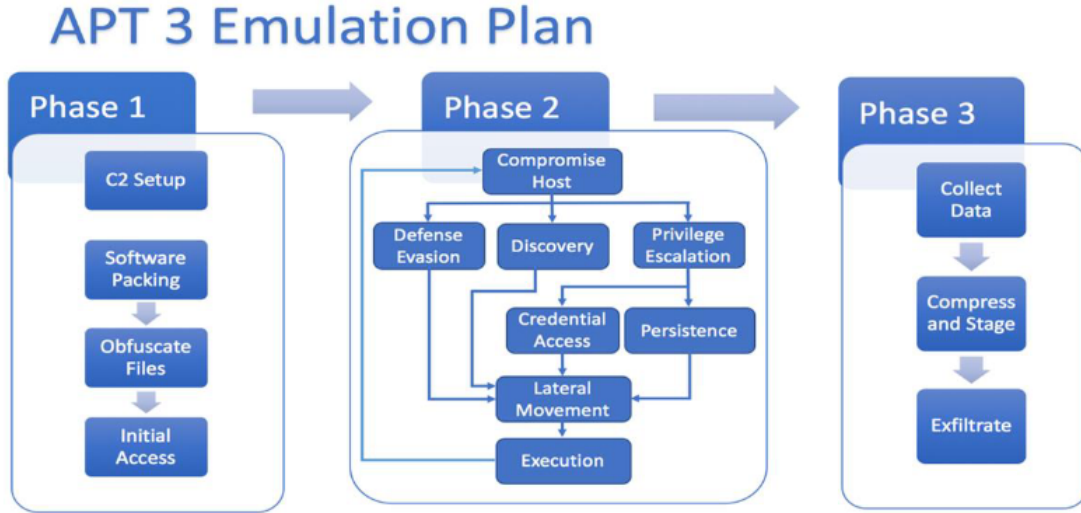


Figure 2.6 – Overview of APT3 emulation plan (taken directly from Pierluigi Paganini, *APT3 Operation Double Tap is targeting recently disclosed Windows vulnerabilities*, 2014, URL: <https://securityaffairs.com/30528/cyber-crime/apt3-operation-double-tap.html> (visited on 09/06/2023)).

scenario do not actually require any specific setup on the architecture. Indeed, at that step the attacker is in full control of the machine anyway, and may perform extraction however they wish. While these techniques could be represented in our formalism (as transitions which lead to no new attack positions or secrets), we chose to ignore them for concision reasons. This also applies to the Discovery and Persistence part of the network propagation step: no specific precautions need to be taken for this technique to be applicable on the architecture, as they just require the attacker to discover what already exists or implement their own tools, respectively. Parts of Phase 1 (such as the Obfuscate Files step or the C2 Setup) were also ignored.

- Once initial access is performed on a machine, the network propagation step can theoretically be performed on any number of machines until the attacker is satisfied. We chose to limit ourselves to 4 machines: one for the initial compromise, and 3 options for lateral movement. This still gives the attacker a choice to make for their next target after initial compromise is achieved. In turn, this could thus let us see which of the machines are more attractive to an attacker, and whether this correlates with their privilege escalation technique or which type of files they contain (in a similar way to Timothy Barron and Nick Nikiforakis, « Picky Attackers: Quantifying

the Role of System Properties on Intruder Behavior », in: *Proceedings of the 33rd Annual Computer Security Applications Conference, ACSAC '17*, Orlando, FL, USA: Association for Computing Machinery, 2017, pp. 387–398, ISBN: 9781450353458, DOI: 10.1145/3134600.3134614, URL: <https://doi.org/10.1145/3134600.3134614>).

For some of these steps, such as Privilege Escalation in the Network Propagation phase, the emulation plan offers more than one technique. For instance, the aforementioned Privilege escalation can be achieved with either T1078: *Valid Accounts*² or T1068: *Exploitation for Privilege Escalation*³. Since some of those steps end up being repeated on several machines (for instance every part of the Network Propagation phase), we tried whenever possible to offer different techniques for each of those steps on each machine.

Given all this, we will now present our formalized attack scenario inspired by the APT3 emulation plan. This attack scenario is played out on 4 machines, named *Bear*, *Raccoon*, *Badger* and *Skunk*. Each machine has 2 users, with one of them being a *SuperUser* with elevated privileges. There is also an additional node (*Attacker*, *SuperUser*)⁴ representing the starting position of the attacker. The goal of the attacker is to reach the winning position (*Skunk*, *root*).

At a technical level, this attack scenario can be described by the graph $\mathbf{S}_e^{\text{tech}} = (\mathbf{P}_e, \mathbf{A}_e^{\text{tech}})$ where $\mathbf{P}_e = (p_i)_{i \in [0,8]}$, $\mathbf{A}_e^{\text{tech}} = (\tau_i)_{i \in [0,15]}$, $\mathbf{S}_e = \{p_0\}$ and $\mathbf{W}_e = \{p_8\}$. A graphical representation of this scenario detailing the values of $(p_i)_{i \in [0,8]}$ and $(\tau_i)_{i \in [0,15]}$ is available in figure 2.7 where the notation Ti refers to the official MITRE numbering for techniques. The attacker has 2 main paths from the starting position to the winning one. He must always start by gaining initial access to machine *Bear*, perform a Privilege Escalation technique to acquire SuperUser privileges, then harvest secrets available on the machine. Then he may go directly to machine *Skunk* and perform T1053: *Scheduled Task/Job* to get to the winning position (*Skunk*, *Superuser*), or he may go to machine *Raccoon* first to acquire the credentials necessary to perform T1078: *Valid Accounts* to achieve the same goal: the winning position (*Skunk*, *Superuser*). It has to be noted that machine *Badger* is unnecessary for both of these winning attack paths here, acting as a dead end to diversify the paths an attacker may explore.

The presence of machine *Badger*, which acts as a mere distraction for the attacker, increases the number of winning attack paths. We will thus ignore it for the purpose of

2. Formerly Legitimate Credentials

3. Formerly Exploit Vulnerability

4. We throughout this work will refer to the attacker's machine as Attacker and consider they have complete control of it.

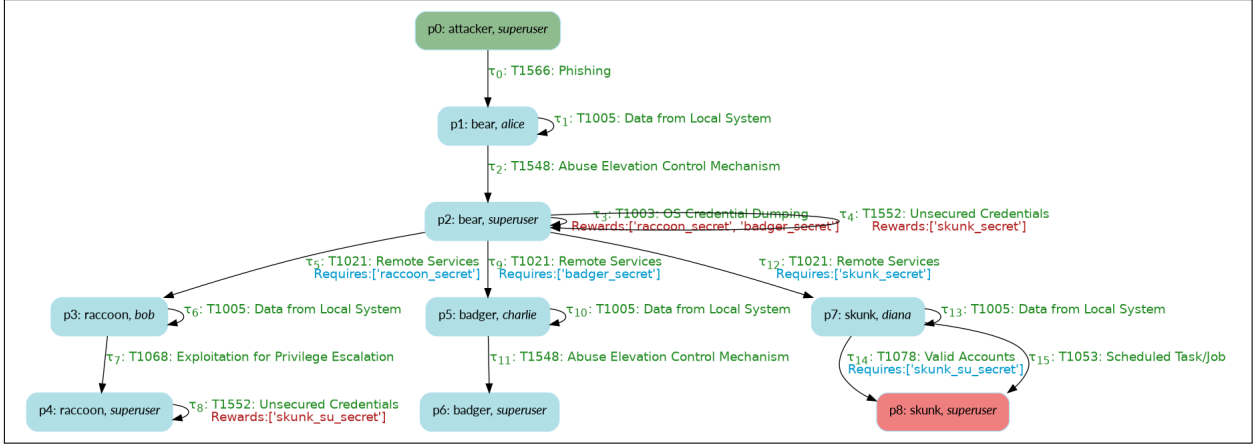


Figure 2.7 – Formalized example scenario on a technical level.

attack path analysis. We also chose to keep some of the T1005: *Data from Local System* that were present in the original report. While these techniques do not provide attackers with secrets or new positions in our modelization, they will be useful for later operations in this thesis as a way to increase the credibility of the architecture (by populating it with files).

We identified two main winning attack paths for the attacker earlier: one where they are required to access the credential in machine *Raccoon*, and one where they directly perform the privilege escalation technique in machine *Skunk*.

Figure 2.8 and 2.9 showcase these two winning attack paths. Note that these paths require the attacker to perform different techniques. As such, an attacker which does not have T1053: *Scheduled Task/Job* in their arsenal will be required to acquire the credentials in machine *raccoon*. These paths are also of different length, with one requiring the attacker to perform at least 3 more techniques. In general, even though each machine follows a similar pattern (using techniques from tactics Lateral Movement, Privilege Escalation then Credential Access), we made sure to vary the techniques available on each of them.

The formalization of this whole scenario, alongside the data structures necessary to formalize any scenario was implemented in Python, and was the first step in the development of URSID. This implementation is able to generate graphical representations of scenarios based on a python or json description. As part of this implementation, there was a need for an exhaustive list of technique names mapped to their number according to MITRE. This list was generated using Hember *et al.* Hemberg et al., *op. cit.* BRON project, which aimed to connect all of MITRE’s publicly available databases together.

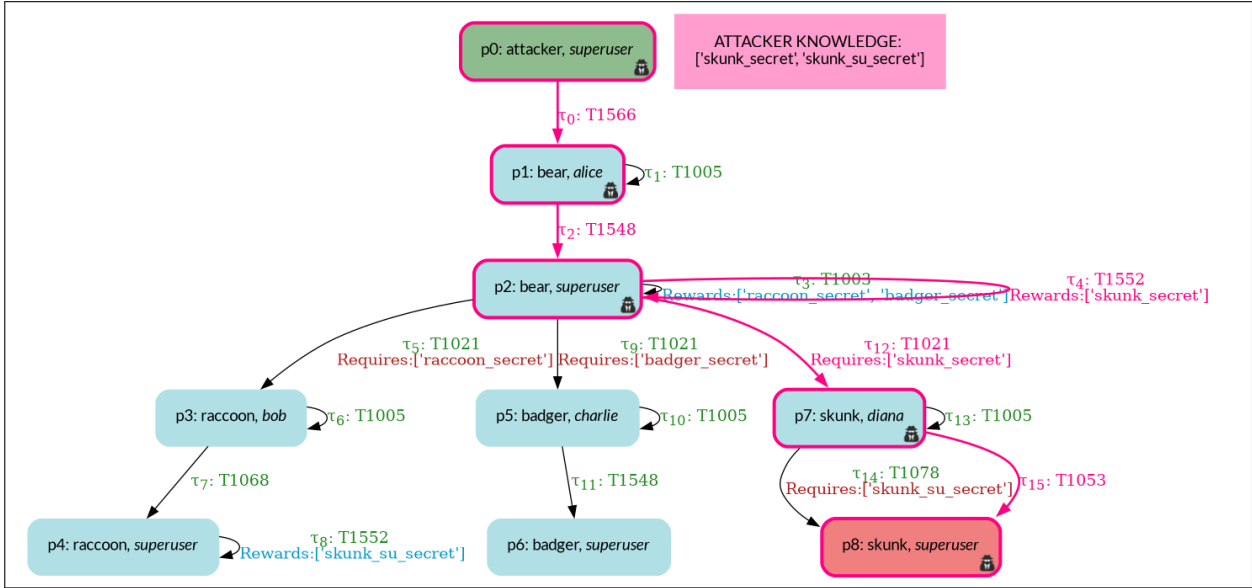


Figure 2.8 – Direct winning attack path.

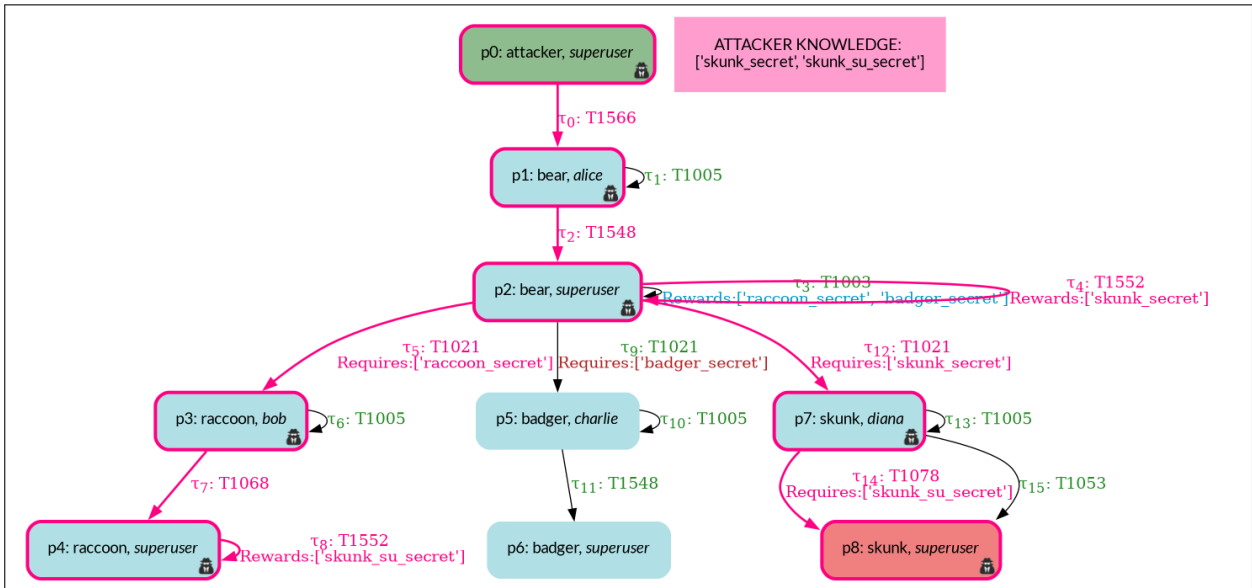


Figure 2.9 – Alternative winning attack path.

In order to make it easier to generate new scenarios, a Graphical User Interface (GUI) was also developed, making it easier to add nodes and transitions between those nodes before saving the result as a json file. Figure 2.10 offers an overview of the GUI, including the base window and the menus to add nodes or transitions. This GUI implementation -

as well as the rest of URSID formalism - is available online BESSON, *op. cit.*, alongside documentation and usage instructions.

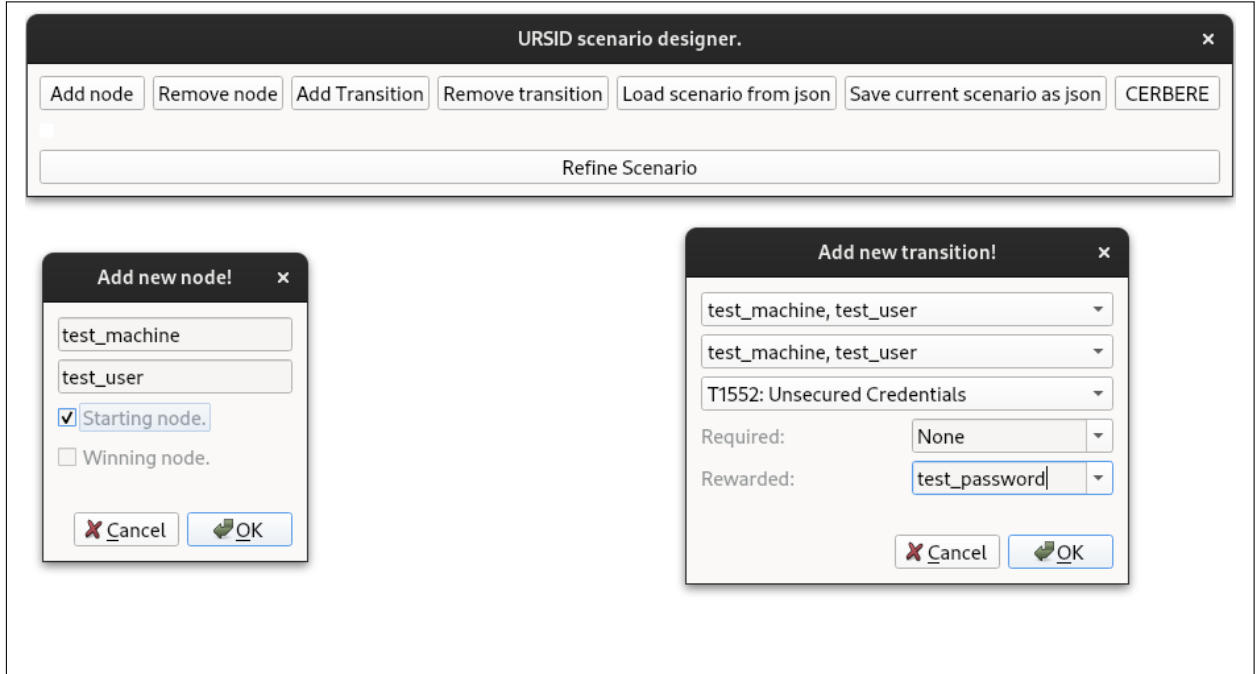


Figure 2.10 – URSID scenario designer GUI.

2.3.1 Conclusion.

This section dealt with our approach to the formalization of attack scenarios in a way that answers our main gripes with the current state of the art of cyber range and vulnerable architecture deployment, mainly a lack of flexibility and re-usability. This included a detailed explanation of the formalism itself alongside showcasing an example scenario inspired by a real life threat actor's modus operandi. This scenario was a starting point in order to showcase the benefits of URSID's formalism, including:

- The ability to present attackers with different winning paths and calculate these paths.
- The ability to model a scenario on a high-level thanks to the use of MITRE techniques.
- The relative ease of tweaking an existing scenario by adding nodes or transitions.

There is however an evident need for a conversion process between a high-level technical level description into a procedural low-level one: otherwise there is no point to this

distinction. This process will also showcase what we believe to be URSID’s formalism best strength, which is these 2 levels of description without affecting the graph structure below. Section 3 will explain how URSID tackles this conversion process by picking procedures for each technique within the scenario, while making sure the overall architecture remains consistent. We will now explain how this process, which we dubbed **procedural refinement**, is achieved and allows URSID to fulfill its design goals.

PROCEDURAL REFINEMENT.

As seen in the previous chapter, scenarios can be described at two different levels: the technical level and the procedural level. The technical level uses MITRE techniques to label its transitions. It is intentionally a high-level description, so that it is possible, for example, to design a scenario that follows the modus operandi of a known attacker without having to worry about implementation choices. Meanwhile, the procedural level is a lower level of description, delving into the details of the actions the attacker must perform to make this transition. This loses some of the genericity provided by the technical level description, but is necessary for the later implementation step.

According to the MITRE TTP nomenclature, a single technique may correspond to a wide range of procedures, without an exhaustive list of procedures per technique. Thus, for a given transition of a scenario described at the technical level, there is a wide range of transitions described at the procedural level in which the chosen procedure corresponds to that technique. Furthermore, a scenario is likely to have more than one transition, and transitions within that scenario are likely to be associated with different techniques. Thus, moving from a technical level description to a procedural level description requires the scenario designer to make many choices, as a single technical level scenario may correspond to a variety of procedural level scenarios.

Figure 3.1 showcases this conundrum, as well as the relationship between techniques and procedures. For instance, a scenario may have a transition labeled with the technique T1068: *Exploitation for Privilege Escalation*. This technique may correspond to a large range of procedures: for instance the exploitation of any CVE leading to a privilege escalation may fit the bill. The exploitation of CVE-2016-5195 NIST, *CVE-2016-5195 Detail*, 2016, URL: <https://nvd.nist.gov/vuln/detail/cve-2016-5195> (visited on 04/20/2024) - in which an attacker abuses a race condition in older versions of the Linux kernel - or of CVE CVE-2017-0005 NIST, *CVE-2017-0005 Detail*, 2017, URL: <https://nvd.nist.gov/vuln/detail/CVE-2017-0005> (visited on 04/20/2024) - in which the attacker makes use of a flaw within the Windows Graphics Device Interface -

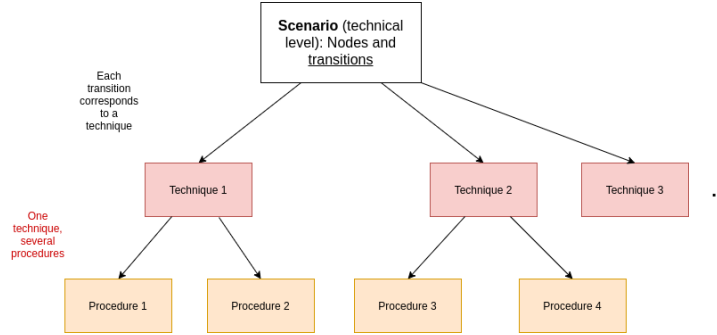


Figure 3.1 – Relationship between scenarios, techniques and procedures.

are two valid procedures for this technique. Choosing one procedure over the other would lead to different procedural level scenarios, which would in the next step lead to different architectures being deployed.

Two steps are thus required for this conversion between technical and procedural level. First, one must have a list of procedures for each technique within the scenario, and then have a way to choose which procedure to pick for each of those techniques, as for reasons we will mention in this chapter these choices cannot just be made at random. The next two sections of this chapter will each detail a part of this process as well, the challenges involved and URSID’s answer to those issues. We will finally conclude this chapter with a short example showing how the APT-3 inspired scenario fares against our process of converting a technical level scenario into several procedural ones, a process which we dubbed **procedural refinement**. A summary of this process was explained and published in a paper for FPS 2023 Pierre-Victor et al., *op. cit.* However, this section goes into much more details about each of those aspects.

3.1 Procedures and constraints.

3.1.1 Problem statement.

The first challenge that comes with defining a scenario on a procedural level is having a consistent way of naming and describing procedures. This is due to the fact that unlike techniques, there is by nature no exhaustive list of procedures as that would require cataloging every single way of compromising a system. The Common Vulnerabilities and Exposures (CVE), a compendium of exploits also maintained by MITRE MITRE, *CVE*

Program Mission, URL: <https://www.cve.org/> (visited on 04/20/2024), does provide a list of software and hardware vulnerabilities, which may then be mapped to techniques using projects such as the work of Hember *et al.* Hemberg et al., *op. cit.* However, plenty of procedures have nothing to do with software vulnerabilities, and are instead the results of an attacker performing legitimate actions on a system to their advantage. For instance, procedures related to T1021: *Remote Services*, in which the attacker logs onto a service accepting remote connections, may just be the attacker logging in normally using passwords they acquired beforehand. Such a procedure is not linked to a CVE, but is nonetheless part of an attacker's path. Even in the case of less legitimate actions - such as procedures related to T1110: *Brute Force* in which the attacker tries to guess a password by attempting multiple login attempts - these procedures may not require a direct exploit of the system per se.

Therefore, the list of techniques considered in the work of this thesis - and ultimately used in URSID - had to be manually selected and implemented. This is an ongoing effort, and new procedures have been added during the course of this thesis to expand the range of techniques covered by URSID, or due to specific experimental needs. This is an unfortunate obstacle to our goal of automating scenario deployment, and arguably the main reason for the state of the art's preference for low-level descriptions. However, due to the way scenarios are designed at a technical level before being automatically transformed to a procedural level, once a procedure is added to the list, it can be reused in any scenario. In other words, this work is an additional burden on URSID developers, but not on scenario designers. Procedure names are not canonical, but have been chosen to succinctly describe how they work and what they do. For example, a procedure corresponding to T1021: *Remote Services* is "Weak SSH Password", where a machine is remotely accessible via SSH with an easy-to-guess password.

The theoretical work of adding procedures to techniques can be done in two different ways. In some cases, we started with the name of a technique and found procedures corresponding to it in a top-down approach, using resources such as MITRE itself or attack reports to find examples. In other cases, we had a procedure that we wanted to implement (for example, a specific web-based exploit) and then worked backwards to figure out what technique it belonged to.

Procedures are therefore referred to by name throughout this thesis. However, it is necessary to associate this name with information about the configuration of the resulting architecture. For example, to deploy an architecture vulnerable to CVE-2016-5195 NIST,

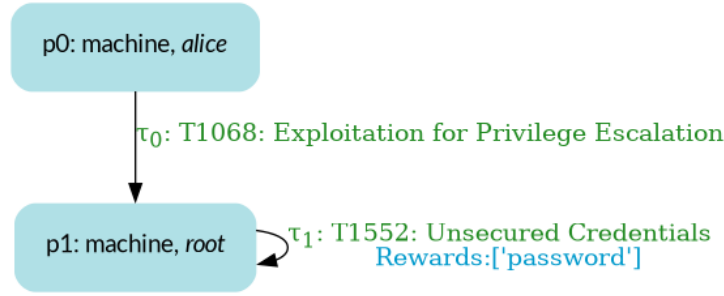


Figure 3.2 – Example scenario on a technical level which will help showcasing architectural inconsistencies.

CVE-2016-5195 Detail, we need to deploy a Linux system with a specific kernel version. This information must be provided at some point in the deployment process. In theory, this step could be done later, for example, by having a separate layer of software with a database that converts procedure names into deployment information. Instead, we decided to directly associate procedure names with all the information needed to deploy an architecture that is vulnerable to them. This is because of another challenge that comes with selecting procedures for a scenario, which we'll call **architectural inconsistencies**.

In order to illustrate the issue, we'll take a simple scenario (shown in figure 3.2) as an example. It consists of a single machine with two attack positions. The attacker starts from the position (*Machine, Alice*), and is expected to perform T1068: *Exploitation for Privilege Escalation* in order to gain access to (*Machine, root*). Once this position is acquired, they are now able to perform T1552: *Unsecured Credentials* in order to gain the secret "password".

Now let's say the procedures we have available for T1068 here are:

- Exploit CVE-2016-5195 *ibid.*, a Linux kernel race condition that affects versions prior to 4.8.3 (π_1).
- Exploit CVE-2017-0005 *idem*, *CVE-2017-0005 Detail*, a vulnerability in a Windows graphical component in specific Windows versions (π_2).

Then, the procedures available for T1552 are:

- Acquire credentials stored in the bash history of a user account in most Linux distributions (which typically saves the last 500 commands entered by a user, potentially saving sensitive data if a user is not careful) (π_3).
- Acquiring credentials saved in a Windows Registry entry, which may sometime store

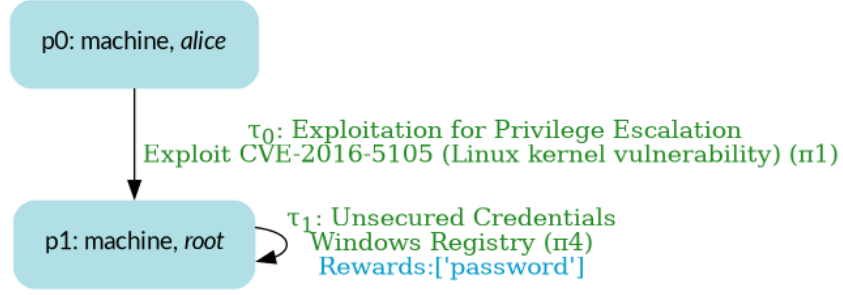


Figure 3.3 – Example scenario on a procedural level presenting architectural inconsistencies.

them for use by other programs or services (π_4).

Procedures π_1 and π_3 are evidently only applicable on a Linux system, while π_2 and π_4 are only applicable for a Windows operating system. Thus, assuming we pick a procedure at random for each technique, we could arrive to a procedural level scenario such as the one in figure 3.3. While both these procedures correspond to their respective technique, it is impossible to deploy an architecture which is vulnerable to both of them¹.

Procedures should therefore include information about how to deploy them, such as operating system requirements (Windows, Linux, which version...), software requirements, *etc.* There is also a need to ensure that two procedures are compatible with each other, i.e. that their requirements can be combined into a meaningful architecture. URSID solves both of these problems at once by introducing the concept of architectural constraints.

Architectural constraints represent all the information needed to implement a procedure, allowing us to a) later deploy it as a virtual machine, and b) check the constraints against each other as we choose our procedures. This includes information about operating systems, software, but also files and account configurations that need to be installed on one or more machines to make them vulnerable to this procedure. We will first examine our choices and representations of constraints.

3.1.2 Architectural constraints.

An **architectural constraint** $\mathcal{C}$ is related to a specific machine m and is denoted by $\mathcal{C}(m)$. A single constraint $\mathcal{C}$ includes Operating System constraints, Account constraints,

1. While very specific operating system configurations or the use of Windows Subsystem for Linux might make it technically possible, we will disregard such cases as niche counter-examples rather than relevant use cases.

Software constraints and File constraints. An OS constraint corresponds to an OS type (such as Windows, Windows server, Ubuntu, Debian, Fedora, *etc.*) and a version or a set of versions for this OS. We consider that a single machine may run only one OS at a time, as virtual machines and containers may be modeled as separate machines. An account constraint details the name, group, privilege, services and credentials of users which have to be added on a machine for the procedure to be available. A software constraint specifies a list of software (name, version and port if applicable) that have to be installed on a machine. These software are installed for every user by default. Finally, a file constraint details information about the files (name, path, permissions, content) that have to be added or modified on a machine. This information includes the path of the file on the resulting machine, the permissions associated to this file and its content. This leads us to definition 3.1.1:

Definition 3.1.1 (Architectural Constraints). Given a machine m , we define an architectural constraint $C(m)$ related to that machine as

$$C(m) = \begin{cases} OS & type, version \\ Software & \text{list of } (software, version, port) \\ Account & \text{list of } (name, group, privilege, services, \\ & credentials) \\ Files & \text{list of } (path, permissions, content) \end{cases}$$

This choice of constraint categories started with an initial reflection on what kind of issues the implementation of a procedure could lead to, and then trial and error iterations on the design as more and more procedures were implemented. For instance, the *services* subconstraint for the *Account* constraint was not initially part of the design, but was added as it became necessary to represent web or database services required for some procedures.

Given two different constraints $C^1(m)$ and $C^2(m)$ (linked to the same machine m), we say these two constraints are compatible if both of these procedures may be implemented on the machine without it leading to the previously mentioned architectural inconsistencies. That is to say, the constraint resulting from the combination of these two constraints is satisfiable. Throughout this thesis, we will represent the concept of constraint combination as $C^1(m) \wedge C^2(m)$. This type of combination is applicable to all four different kind of constraints (OS, Software, Account, Files). The specifics of these combinations depend on

the constraint (and its subconstraints), but 2 categories emerge:

- Constraints which can be combined with each other by simply adding them together. For instance, if a procedure must install the sudo software in order to be exploited, while another must install the nmap software, we know that the resulting architecture must install both sudo and nmap. This type of combination is represented with the $\wedge$ operator.
- Constraints which have to be fused with each other. For instance, in the case of OS constraints, a machine can only run a single operating system at a time. Therefore, if one procedure must be on Debian with a version ≥ 9.0 and another must be on Debian with any version, both these constraints can be combined into the most restrictive constraint of Debian with version ≥ 9.0 . We introduce the $\bowtie$ operator in order to represent these combinations, in which two subconstraints are fused into one.

Formally, for two subconstraints Sc_1 and Sc_2 , $Sc_1 \wedge Sc_2$ returns a new subconstraint Sc_3 , which can be either $\perp$, which is the unsatisfiable subconstraint and a conclusion of incompatibility, or a valid combination of Sc_1 and Sc_2 . Of course, the details of this combination depend on the nature of the operands. Formally, there is a different $\bowtie$ operator for each type of subconstraint. For the sake of clarity, we choose to use the same symbol, while always making sure that both operands are of the same type. Intuitively, two constraints are compatible if their OS, Software, Account and Files constraints are all compatible with each other. We will refer to an OS constraint associated to a given architecture constraint $\mathcal{C}(m)$ as $\mathcal{C}_{OS}(m)$, with the same for software, account and files. Finally, when we need to differentiate two different constraints $C(m)$, we may do so by numbering them $C^1(m)$ and $C^2(m)$.

This leads us to definition 3.1.2:

Definition 3.1.2 (Architectural constraints). For two constraints $C^1(m)$ and $C^2(m)$, $C^1(m) \wedge C^2(m) \neq \perp$ if and only if:

- $\mathcal{C}_{OS}^1(m) \wedge \mathcal{C}_{OS}^2(m) \neq \perp$
- $\mathcal{C}_{soft}^1(m) \wedge \mathcal{C}_{soft}^2(m) \neq \perp$
- $\mathcal{C}_{account}^1(m) \wedge \mathcal{C}_{account}^2(m) \neq \perp$
- $\mathcal{C}_{files}^1(m) \wedge \mathcal{C}_{files}^2(m) \neq \perp$

According to the MITRE nomenclature, each technique may be associated with multiple procedures. We may define the function α such that for each technique τ , $\alpha(\tau) = \{pi_i\}$

the set of procedures corresponding to this technique. Thus, for a given transition $t = (p_1, p_2, (\tau, \mathbf{pre}, \mathbf{post}))$, with $p_1 = (m_1, u_1)$ and $p_2 = (m_2, u_2)$ labeled on the technical level, a procedure $p_i \in \alpha(\tau)$ may be used to label this transition on a procedural level. A single procedure can thus induce constraints on up to two different attack positions p_1 and p_2 , as the choice of a procedure may have consequences on either the entry or exit position of the transition. For instance, the previously mentioned procedure related to T1021: *Remote Services*, "Weak SSH Password" (in which a machine is accessible remotely through SSH with an easy to guess password) has consequences on the exit attack position: it must be accessible through a SSH service, it must be accepting passwords, that password must be weak, *etc.* In practice, most procedures only induce constraints onto the exit position. However, some cases benefit from having the option of adding entry constraints. Some procedures may be easier to accomplish for the attacker if they have sufficient permissions on their current position in order to install their preferred software, or software may be installed beforehand to facilitate the work of the attacker. For instance, the previous procedure "Weak SSH Password" may also have an entry constraint requiring the entry position to have a network scanning software (such as nmap Gordon Lyon, *Nmap*, 1997, URL: <https://nmap.org/> (visited on 08/30/2023)) installed.

A difficulty score is also assigned to each procedure to approximate the difficulty level of each generated scenario by summing the scores of each procedure involved. This difficulty score is chosen by the scenario designer when implementing the procedures. This can be used to provide additional data to defenders, e.g. to study the correlation between the difficulty of an instance and the speed at which it is compromised, or to provide attackers with a greater variety of challenges if their skills are heterogeneous.

In addition to its architectural constraints, a procedure π may have π_{pre} preconditions, i.e. external knowledge required by the attacker to execute the procedure successfully. Similarly, the execution of a procedure may provide the attacker with additional knowledge we refer to as scenario secrets, which he needs to discover to advance. These secrets can be identifiers, IP addresses, machine names, the contents of certain files and so on. Each procedure therefore has its own list of required and rewarded secrets that corresponds to the **pre** and **post** defined at the technical level. These indicate the type (password, ssh keys...) and amount of secret which may be either required or rewarded by the procedure. During the refinement process, these preconditions are checked against the **pre** and **post** of the transition itself for a potential match.

In order to illustrate this concept, let's take two procedures from URSID:

- π_1 : *Remote Services: SSH access from password*, which corresponds to an attacker getting access to an account through SSH thanks to a password they acquired beforehand. This procedure's secret preconditions are:
 - "Requires": [{"Type": "PLAINTEXT_PASSWORD", "Amount": "1"}]
 - "Rewards": [].
- π_2 : *Remote Services: Weak SSH password*, which corresponds to an attacker getting access to an account through SSH by brute-forcing its password. This procedure has no secret preconditions.

Intuitively, π_1 would be more applicable to a transition which requires a secret, while π_2 would prefer a transition with no such requirements. If we were to associate π_2 with a transition that requires secrets, such requirements would not be needed in practice by the attacker, which defeats the purpose of the scenario. On the other hand, if we were to associate π_1 with a transition with no requirements, the resulting scenario might be unbeatable in practice, as the attacker would have no way of acquiring the required credential beforehand. This is why we include the amount of secrets potentially provided or required by a procedure in its secret preconditions. As for secret types, they are here to make sure that there is consistency within the type. For instance, if a procedure rewards a SSH private key as a secret, it would make no sense for that reward to be used as a requirement for a password-related procedure.

Two subconstraints (*Account.credentials* and *Files.content*) may interact with secrets. These correspond to setting account passwords or keys to specific values and leaking secrets as content of files, respectively. These subconstraints specify whether they interact with secrets, and if they do the type and amount of secrets they may deal with. By combining this information with the secret preconditions, we are able to partition secrets between the constraints, i.e decide which constraint deals with which secret in the event a transition requires or rewards more than one secret. Formally:

Definition 3.1.3 (Secret preconditions). For a given procedure π , we define its secret preconditions as $\pi_{secrets} = \{\mathbf{pre} : [type_1, amount_1, \dots]; \mathbf{post} : [type_2, amount_2, \dots]\}$ where:

- $type_i$ corresponds to a type of secret (PLAINTEXT_PASSWORD, SSH_KEYS, etc.).
- $amount_i$ indicates how many instances of that secret may be required or rewarded by the procedures (can be the wildcard "*" if any non-zero amount is valid).

We will now cover each of the different constraints alongside their subconstraints. This

will include a discussion on how to represent these subconstraints and how they were implemented into URSID.

Operating System constraints Operating system constraints indicate that a procedure requires a specific operating system or range of operating systems to be applicable. This is used to indicate both the type of operating system (Windows, Ubuntu, Debian...) and the valid versions that are vulnerable to this procedure. This is relevant in many cases, as exploits tend to be patched out of updated versions.

We make use of the MITRE Common Platform Enumeration (CPE) to name operating systems when applicable. Defined as "structured naming scheme for information technology systems, software, and packages", the CPE format provides an agreed upon list of names for a wide array of software and operating systems. An entry in the Official CPE Dictionary NVD, *Search Common Platform Enumerations*, 2009, URL: <https://nvd.nist.gov/products/cpe/search> (visited on 04/30/2024) is formatted as follows:

`cpe:/<part>:<vendor>:<product>:<version>:<update>:<edition>:<language>`

In which `<part>` is either `a` (for applications), `o` (for operating systems) or `h` (for hardware), and all other entries are fairly self-explanatory. For instance, the CPE corresponding to a Debian 11.0 operating system would be `cpe:o:debian:debian_linux:11.0:*:*:*:*:*`, with `*` referring to a wildcard character (meaning any value is acceptable). In practice, the `<update>`, `<edition>` and `<language>` tags were almost always unnecessary for our purposes, and we handle `<version>` and `<part>` separately. Our use of the CPE format limits itself to the `<vendor>:<product>` field, which will be used to name operating systems and software when available. While this format is not perfect - in particular in regard to the limits of its exhaustivity -, we consider it to be a helpful basis for naming operating systems and software, coming up with our own names in case the format falls short. Furthermore, the fact it was originally designed by MITRE make it consistent with our use of MITRE TTPs as a basis for our formalism, and may occasionally offer some links between the two nomenclatures. For instance, each CVE (whose exploitation by an attacker may correspond to a procedure) is associated with a list of software and operating systems under the CPE format.

Formally, this leads us to definition 3.1.4:

Definition 3.1.4 (OS constraints). An operating system constraint $C_{OS}(m)$ is composed of two subconstraints *type* and *version*, where:

- *type* is the name of the OS, written as `<vendor>:<product>` according to the CPE nomenclature when available. For instance, `debian : debian_linux` refers to an operating system running a Debian Linux distribution.
- *version* is a range of valid versions written as float intervals. For instance, `== 9.0` means the operating system has to be at version 9.0 exactly, while `>= 20.04` means the version has to be at least 20.04. In the case where any version is valid, the wildcard character `"*"` can be used.

Due to a lack of standardized format for version numbers², converting versions to floats back and forth can be tricky, such as when for instance version 3.10 is an update to version 3.9. These cases have to be handled manually when comparing versions, but are fortunately pretty rare.

Note that in some cases, an OS constraint is not tied to a specific OS distribution, but rather to a broad class of operating systems. For example, a procedure related to the use of the `sudo` package is inherently a Linux exclusive, but may apply to both Ubuntu and Debian distributions. This will be relevant when discussing incompatibilities between constraints.

Software constraints Software constraints indicate that a procedure requires a specific software to be installed in order to be applicable. They end up being fairly similar to operating system constraints, requiring a name (making use of the CPE format again when possible) and a version. Some software - for instance web applications - may however require network ports in order to function. We consider that a single port may not be shared by two different applications³, which lead us to a third optional `port` subconstraint to software constraints. Formally, this leads us to definition 3.1.5:

Definition 3.1.5 (Software constraints). A software constraint $C_{soft}(m)$ is composed of two subconstraints *type* and *versions* and an optional subconstraint *port*, where:

- *type* is the name of the software, written as `<vendor>:<product>` according to the CPE nomenclature when available. For instance, `sudo_project:sudo` refers to the `sudo` software, used on Linux operating systems to delegate authority.

2. Options exist for a given software (for instance Python package versions), but we are here attempting to compare versions of any possible software.

3. Some software may co-exist on a single port in theory, but we consider those to be niche cases.

- *version* is a range of valid versions written as intervals. For instance, `== 1.8.27` means the software has to be at version 1.8.27 exactly. In the case where any version is valid, the wildcard character `*` can be used.

Account constraints Account constraints indicate that a procedure requires specific user accounts configuration. This encompasses the name and user group of the account, but also its privileges, services and credentials. As explained in definition 3.1.1, an account constraint $C_{account}(m)$ is a list of *(name, group, privilege, services, credentials)* subconstraints. These subconstraints work as follows:

- *name* is the name of the account affected by the constraint. This can be a fixed value, for cases when the procedure needs to create a new account with a specific name, or if the procedure changes the configuration of an account available by default on the platform (such as *root* on Linux systems). Most of the time however procedures attempt to change the configuration of the account corresponding to one of the attack positions linked to the transition. In order to account for these cases, *name* can take the values `ENTRY_USER` or `EXIT_USER`, which correspond to the user of the entry attack position or the exit attack position of the transition respectively. These will be dynamically replaced by the appropriate value when refining the scenario.
- *group* works identically to *name*. In the current implementation of URSID, every procedure picks identical values for *group* and *name*, but separating them gives us a greater amount of flexibility for future cases.
- *privilege* represents the user's level of access on the machine. For now this can take two values: "User" (no particular access rights), and "SuperUser" (the account is given administrative privileges). This is particularly useful for procedures related to privilege escalation, as the procedure itself can now specify that the exit attack position will have increased privileges. These two options may be extended in the future, for instance when we tackle the deployment of Windows systems.
- *services* represents services (for instance web, database, ssh, etc.) which will have to be launched and owned by this specific user. This was a late addition into URSID's design, as this was originally handled by software constraints. This became an issue when implementing procedures linked to T1021: *Remote Services*, as these typically allow an attacker access to an account which launched a vulnerable service. In these cases, it is necessary to specify which account will be given access to. *services* entries correspond to the name of the service that has to be launched.

- *credentials* allows the procedure to specify values for passwords or keys related to the user account. In URSID's current implementation this always refers to the user's account password, but other types of credentials (such as ones related to services) may be possible in the future. This is one of the subconstraints that are affected by secret preconditions, and is as such more complicated than the others, being comprised of sub-fields:
 - "Credentials Type" refers to the type of credential that has to be generated. For instance, this can be a randomized but easy to guess value (`WEAK_SSH_PASSWORD`) or a plaintext password in general (`PLAINTEXT_PASSWORD`).
 - "Requires Secret" indicates whether the credential depends on a secret or not. This is relevant for cases where the attacker is supposed to get access to an account by reusing their knowledge acquired somewhere else within the scenario. On the contrary, some credential subconstraints (such as the previously mentioned `WEAK_SSH_PASSWORD`), are designed to be guessed by the attacker directly. In the event where "Requires Secret" is true, two additional fields are required in order to account for secret partition within the procedure:
 - (Only if "Requires Secret" is true) "Secret Type" indicates the kind of secret which is required by the procedure. For instance, this may take the value `RANDOM_WEAK_PASSWORD` in order to indicate that this will be a randomly generated weak (as in, easy to brute force) unencrypted password.
 - (Only if "Requires Secret" is true) "Secret Amount" indicates how many secrets of the given type are required by the procedure. This can be a wildcard in the case where the amount does not matter.

Files constraints File constraints indicate that a procedure requires specific actions to be taken on the machine's file system. For instance, a procedure may require existing files to be edited (such as configuration files), new files to be added (such as databases), permissions of files to be changed or existing files to be destroyed. Note that the use of the word "file" here may refer to directories as well. Files subconstraints are as follows:

Definition 3.1.6 (Files constraints). A files constraint $C_{files}(m)$ is composed of three subconstraints *path*, *perm* and *content*, where:

- *path* refers to the path of the file or directory on the machine. Files are written using the Linux format (using the "/" character as opposed to Windows "\\"). If a path end

with a "/", it points to the final location of the file. If not, it points to the complete name of the file.

- *perm* refers to the permissions of the file, including its owner, its group, and its access permissions written in the Linux octal notation. These indicate the read, write and execute permissions for the owner of the file, group, and others. While this format is native to the Linux operating system, similar concepts exist in Windows.
- *content* refers to the nature of the file modification itself. This may indicate that a file needs to be created (in which case its content will also be specified), that an existing file needs to be appended or edited, or that the file corresponding to the path needs to be deleted. Similarly to *account.credentials*, *content* comprises several fields:
 - "Content Type" is a string referring to the type of file and edit that needs to be performed, for instance "BASH_HISTORY_PASSWORD_LEAK".
 - "Requires Secret" indicates whether the file depends on a secret or not, relevant for cases where credentials are stored or leaked within files. In the event where "Requires Secret" is true, two additional fields are required in order to account for secret partition within the procedure:
 - (Only if "Requires Secret" is true) "Secret Type" indicates the kind of secret which is required by the procedure.
 - (Only if "Requires Secret" is true) "Secret Amount" indicates how many secrets of the given type are required by the procedure. This can be a wildcard in the case where the amount does not matter.

Figure 3.4 showcases how architectural constraints fit into URSID's workflow, and how procedures are translated into sets of constraints.

Having defined these, we will now discuss the issue of constraint compatibility. This is done to solve our problems of architectural inconsistencies that could arise from a careless choice of procedures for each technique - for example, a procedure that requires Windows and a procedure that requires Linux to coexist on the same machine. In the next section, we detail our choice of constraint incompatibilities for each of our categories (OS, Software, Account, Files), and then present how we use them to refine a scenario through a backtracking process. The output of this refinement process will be a scenario described on a procedural level, where each transition is labeled with a procedure corresponding to the technique that was present on a technical level, and which does not have any

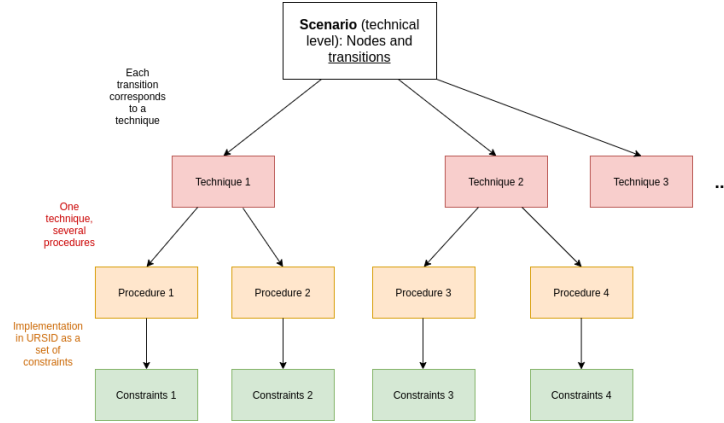


Figure 3.4 – Relationship between procedures and constraints as part of URSID’s workflow.

incompatibilities between procedures.

3.2 Backtracking refinement algorithm.

3.2.1 Constraint incompatibilities.

Constraint incompatibilities are handled by constraint category. That is, given two architectural constraints, incompatibilities between them are handled by checking for incompatibilities between their OS, Software, Account, and Files constraints. This in turn involves checking for incompatibilities between their associated sub-constraints, but the details of this operation depend on the constraint involved.

OS constraints Intuitively, two OS constraints are compatible if both their type and versions subconstraints are compatible with each other, as a machine may only run a single OS. Formally, this leads us to definition 3.2.1:

Definition 3.2.1 (OS constraint combinations). Given two operating system constraints associated to a machine m $C_{OS}^1(m) = \{type_1, version_1\}$ and $C_{OS}^2(m) = \{type_2, version_2\}$, $C_{OS}^1(m) \wedge C_{OS}^2(m) = C_{OS}^3(m) = \{type_3, version_3\}$ where

- $type_3 = type_1 \bowtie type_2$, the operating system type that satisfies both of these subconstraints.
- $version_3 = version_1 \bowtie version_2$ the interval of versions that satisfy both $version_1$ and $version_2$.

For instance, using examples taken directly from procedures implemented in URSID:

- $C_{OS}^1(m) = \{\text{Type: "canonical:ubuntu_linux", Version: "=="20.04"}\}$ requires the machine to be on version 20.04 of the Ubuntu Linux distribution.
- $C_{OS}^2(m) = \{\text{Type: "debian:debian_linux", Version: "*"}\}$ requires the machine to be on any version of the Debian Linux distribution.
- $C_{OS}^3(m) = \{\text{Type: "linux", Version: "*"}\}$ requires the machine to be on any version of any Linux distribution.

The resulting possible combinations of constraints are:

- $C_{OS}^1(m) \wedge C_{OS}^3(m) = \{\text{Type: "canonical:ubuntu_linux", Version: "=="20.04"}\}$
- $C_{OS}^2(m) \wedge C_{OS}^3(m) = \{\text{Type: "debian:debian_linux", Version: "*"}\}$
- $C_{OS}^1(m) \wedge C_{OS}^2(m) = \perp$, as a machine cannot be both on Ubuntu and Debian.

Software constraints Unlike operating system however, several different pieces of software may coexist on the same machine. Intuitively, two software constraints that refer to two different pieces of software can be combined directly, as long as they do not use the same port. In the case where two software constraints refer to the same software, their versions are combined similarly to operating systems. Formally:

Definition 3.2.2 (Software constraint combinations). Given two software constraints $C_{soft}^1(m) = (type_1, version_1, port_1)$ and $C_{soft}^2(m) = (type_2, version_2, port_2)$, $C_{soft}^1(m) \wedge C_{soft}^2(m) = C_{soft}^3(m) = (type_3, version_3, port_3) =$

- $(type_1, version_1, port_1) \wedge (type_2, version_2, port_2)$ if $(type_1 \neq type_2 \wedge port_1 \neq port_2)$
- $(type_1, (version_1 \bowtie version_2), (port_1 \bowtie port_2))$ if $type_1 = type_2$

For instance, using examples taken directly from procedures implemented in URSID:

- $C_{soft}^1(m) = \{\text{Type: "sudo_project:sudo", Version: "=="1.8.27"}\}$ requires the sudo package to be installed at version 1.8.27.
- $C_{soft}^2(m) = \{\text{Type: "npm", Version: "*", Port: 3000}\}$ ⁴ requires the npm package to be installed at any version, which will use port 3000.

Since these 2 constraints share neither a port nor a software type, they can be easily combined into $C_{soft}^1(m) \wedge C_{soft}^2(m)$, indicating that the resulting architecture will need to have both the sudo and npm packages installed.

4. No satisfying CPE could be found for npm at the time, hence the lack of vendor in the name.

Account constraints Intuitively, account constraints do not interact unless they refer to the same account. If they do, the subconstraints for groups, privileges, services, and credentials must be checked for compatibility. These are incompatible if they end up giving different values to the same field, such as one procedure specifying that one user has superuser privileges while the other doesn't. This leads us to combination rules 3.2.3:

Definition 3.2.3 (Account constraint combinations). Given two account constraints $C_{account}^1(m) = (name_1, group_1, priv_1, serv_1, cred_1)$ and $C_{account}^2(m) = (name_1, group_1, priv_1, serv_2, cred_2)$ and $C_{account}^3(m) = (name_3, group_3, priv_3, serv_3, cred_3)$ =

- $(name_1, group_1, priv_1, serv_1, cred_1) \wedge (name_2, group_2, priv_2, serv_2, cred_2)$ if $name_1 \neq name_2$
- $(name_1, (group_1 \bowtie group_2), (priv_1 \bowtie priv_2), (serv_1 \bowtie serv_2), (cred_1 \bowtie cred_2))$ if $name_1 = name_2$

In order to illustrate account constraint combinations, we will take 3 examples from procedures implemented in URSID:

- $C_{account}^1(m) =$
 - "Name": "EXIT_USER"
 - "Group": "*"
 - "Privilege": "SuperUser"
 - "Credentials": {"Credential Type": "*", "Requires Secret": false}
- $C_{account}^2(m) =$
 - "Name": "EXIT_USER"
 - "Group": "*"
 - "Privilege": "*"
 - "Credentials": {"Credential Type": "RANDOM_WEAK_PASSWORD", "Requires Secret": false}
- $C_{account}^3(m) =$
 - "Name": "EXIT_USER"
 - "Group": "*"
 - "Privilege": "*"
 - "Credentials": {"Credential Type": "PLAINTEXT_PASSWORD", "Requires Secret": true, "Secret Type": "PLAINTEXT_PASSWORD", "Secret Amount": "*"}

$C_{account}^1(m)$ indicates that the exit attack position should correspond to a privileged user. $C_{account}^2(m)$ implies that the exit user should have a weak password, while $C_{account}^3(m)$ specifies that this password should correspond to a secret found elsewhere within the scenario. Since all these constraints are related to the same user, we have to check the other subconstraints. Since $C_{account}^1(m)$ only specifies information on the level of privilege, which both $C_{account}^2(m)$ and $C_{account}^3(m)$ leave untouched, $C_{account}^1(m) \wedge C_{account}^2(m)$ and $C_{account}^1(m) \wedge C_{account}^3(m)$ are both valid combinations. However, $C_{account}^2(m) \wedge C_{account}^3(m) = \perp$ as they both try to give different values to the same credential.

Files constraints Similarly to *accounts*, files constraints are compatible with each other as long as they do not interact with the same file path. In the event where they do, the file permissions and content subconstraints have to be compatible. While files permission is fairly straightforward (requiring name, group and permission to match exactly), files content compatibility is on paper a very complicated issue to solve. For instance, if two constraints try to edit the same file, they may or may not be compatible with each other depending on the nature of the edit: do they edit the same line? What if they add contradictory information to a configuration file? We've decided to simplify the problem for URSID, and assume that file content subconstraints are never compatible unless they are exactly the same. While this is not a perfect solution - and alternatives such as using Git as a way to merge two conflicting file content subconstraints were considered - in practice, this edge case has rarely, if ever, arisen in the deployment of URSID infrastructures. This leads us to combination rules 3.2.4:

Definition 3.2.4 (Files constraint combinations). Given two files constraints $C_{files}^1(m) = (path_1, perm_1, content_1)$ and $C_{files}^2(m) = (path_2, perm_2, content_2)$, $C_{files}^1(m) \wedge C_{files}^2(m) = C_{files}^3(m) = (path_3, perm_3, content_3) =$

- $(path_1, (perm_1 \bowtie perm_2), (content_1 \bowtie content_2))$ if $path_1 = path_2$
- $(path_1, perm_1, content_1) \wedge (path_2, perm_2, content_2)$ if $path_1 \neq path_2$

Taking examples from URSID once again:

- $C_{files}^1(m) =$
 - "Path": "~/ .bash_history"⁵
 - "Permissions": {"User": "EXIT_USER", "Group": "EXIT_USER", "Perm": "0600"}

5. ~ refers to the relative path of the file's owner home directory.

-
- "Content": {"Content Type": "BASH_HISTORY_PASSWORD_LEAK",
"Requires Secret": true,
"Secret Type": "PLAINTEXT_PASSWORD", "Secret Amount": "*"}
 - $C_{files}^2(m) =$
 - "Path": "/etc/ssh/sshd_config"
 - "Permissions": {"User": "EXIT_USER", "Group": "EXIT_USER",
"Perm": "0644"}
 - "Content": {"Content Type": "ENABLE_SSH_PASSWORD_CONNECTION",
"Requires Secret": false}
 - $C_{files}^3(m) =$
 - "Path": "~/"
 - "Permissions": {"User": "EXIT_USER", "Group": "EXIT_USER", "Perm": "0755"}
 - "Content": {"Content Type": "UNIX_DOCUMENTS",
"Requires Secret": false}

$C_{files}^1(m)$ leaks a password by adding the value of a secret to the bash history of the user, $C_{files}^2(m)$ tweaks the configuration of the user's SSH service to make sure it accepts password connections, and $C_{files}^3(m)$ just adds documents to the user's home directory in order to make it more credible. Since all of those constraints deal with different files, they are all compatible with each other, thus $C_{files}^1(m) \wedge C_{files}^2(m) \wedge C_{files}^3(m) \neq \perp$.

Limits of this approach. We acknowledge that some limitations have been encountered in the development and deployment of URSID. This is due to the sheer variety of possible architectural configurations available, and our need to reduce this to a description simple enough to be hand-writable. We list some of these potential areas for improvement below:

- All procedures so far that were implemented and tested were for Linux systems. While we tried to make architectural constraints platform-agnostic and believe that every subconstraint would be applicable to a Windows system - even when we make use of the Linux notation for the file system for instance - without a concrete implementation we cannot guarantee that this current choice of subconstraints would not need to be tweaked for such purposes.
- Some of the resolutions of subconstraint compatibilities (the ones making use of the $\bowtie$ operator) are nontrivial, the most prominent one being *files.content*. Our solutions

for those is to make strong hypotheses - such as assuming that two constraints dealing with the same file must be incompatible -, despite those being more strict than necessary - two constraints may edit the same configuration file in a compatible way -. These approximations were done due to a) the difficulty of the problem and b) the relative rarity of such cases appearing, putting them low on our development priority list.

- Some values (such as *account.credentials*, *files.content* and anything related to secrets) are represented as manually chosen strings which do not rely on an existing nomenclature, resulting in a non standardized format.
- Network configuration issues, except for port and services, are overall not accounted for by subconstraints. This was done because the technical level of a scenario also does not account for network configurations, instead leaving that work to the virtual machine deployment software. We will detail how URSID handles networking questions in the next chapter, but taking it more into account in either the technical description or the procedural constraints may have been more efficient in the long run.

We however believe that these choices are sound for a majority of applications and scenarios, and managed to define and deploy two very different scenarios as part of research experiments (detailed in the next chapter) using this formalism.

A summary of every constraint, subconstraints and combination rules can be found in figure 3.5. Having now explained how each of them work, alongside how secret preconditions are handled, we will now delve into how URSID manages to convert a scenario described in a technical level into a scenario on a procedural level, by picking procedures written as a set of architectural constraints.

3.2.2 Procedural refinement.

As a reminder, a procedural level scenario follows the same formalism as a technical level scenario, except that the edge sets are labeled by attack procedures rather than techniques. The constraints generated by the choice of procedures make it possible to specify the architecture if these constraints are not inconsistent among them. We use a **backtracking refinement algorithm** to generate an architecture that can play a scenario described at the procedure level. This algorithm walks through the scenario graph and tries to replace each attack technique that labels an edge in the scenario with an attack

Subconstraint type	Sc_i^{type}	$Sc_1^{type} \wedge Sc_2^{type}$
OS	$type_i, version_i$	$((type_1 \bowtie type_2), (version_1 \bowtie version_2))$
Software	$type_i, version_i, port_i$	$(type_1, version_1, port_1) \wedge (type_2, version_2, port_2)$ if $(type_1 \neq type_2 \wedge port_1 \neq port_2)$ $(type_1, (version_1 \bowtie version_2), (port_1 \bowtie port_2))$ if $type_1 = type_2$
Account	$name_i, group_i, priv_i, serv_i, cred_i$	$(name_1, group_1, priv_1, serv_1, cred_1) \wedge (name_2, group_2, priv_2, serv_2, cred_2)$ if $name_1 \neq name_2$ $(name_1, (group_1 \bowtie group_2), (priv_1 \bowtie priv_2), (serv_1 \bowtie serv_2), (cred_1 \bowtie cred_2))$ if $name_1 = name_2$
Files	$path_i, perm_i, content_i$	$(path_1, (perm_1 \bowtie perm_2), (content_1 \bowtie content_2))$ if $path_1 = path_2$ $(path_1, perm_1, content_1) \wedge (path_2, perm_2, content_2)$ if $path_1 \neq path_2$

Figure 3.5 – Constraint combination rules for 2 procedures

procedure that instantiates the technique, as long as the constraints on the architecture remain coherent.

The rules for adding constraints are the ones we covered in the previous section. These rules allow us to combine constraints together and check whether the result is consistent or not. The set of all constraints over the resulting architecture is represented as a dictionary of constraints C containing entries for every machine m . This dictionary will be filled and edited throughout the course of our backtracking algorithm, and is considered to be unsatisfiable ($C = \perp$) if any of its entries are unsatisfiable. Essentially, each transition within the scenario will be associated with a procedure corresponding to its techniques one by one.

This procedure is chosen at random from the list of all procedures implemented for that technique, and its architectural constraints are combined with the current list of constraints for each machine C . If the resulting combination is unsatisfiable, another procedure from the same technique is chosen. If none of the possible procedures are compatible, we backtrack and reverse our choice of procedure for a previous transition, and try another one instead. In the end, either every transition of the scenario gets a valid procedure, in which case the scenario is considered refined, or every possible choice of procedures ends in incompatibilities. In this case, the scenario is considered unrefineable, and it is necessary to tweak the scenario on a technical level, or add new procedures for some of the techniques.

C is by default empty at first. The list of all machines is gathered from a simple walk

through the technical representation of the scenario, as each attack position contains the name of its machine. This walk is also used in order to create additional account constraints. For each attack position (m, u) is created an account constraint $C_{account}(m) =$

- "Name": u
- "Group": "*"
- "Privilege": "*"
- "Credentials": {"Credential Type": "*", "Requires Secret": false}

This constraint only specifies that machine m will now have a user named u .

C may also be initialized with additional constraints if needed. For instance, if the scenario designer only wants to deploy a specific type of operating systems (such as Linux) or wants software to be installed by default on the machines, they may do so by initializing $C(m)$ with nonempty values for each machine m . Specific machine configurations can also be tweaked if desired. We make use of this option in the current implementation of URSID for several reasons:

- Specify the range of OSes we are able to deploy (for now only specific Linux systems).
- Add default software, such as **rsync** (helps to deploy files), **ufw** (helps with firewall configuration), **ac1** (helps with package management), **netcat** and **telnet** (gives the attack more networking options).
- Add users to a machine without needing to add them to the technical level scenario. This is used in order to increase the credibility of the machine, but these accounts are not part of any modeled attack path.
- Change the login banner and welcome message of a machine.
- Add logging software and configuration to a machine (such as **auditd**).

Having any of these options enabled results in $C(m)$ being initialized with nonempty constraints. These base constraints will then be compared against procedural constraints throughout the refinement process, guaranteeing for instance that only Linux procedures are picked if this option was chosen. Practically, these options are enabled through the use of optional configuration files written in YAML.

Transitions that deal with secrets (i.e. either **pre** or **post** isn't empty) are prioritized. This is due to the fact that secret type incompatibilities were one of the most common reasons why procedures were incompatible with each other. Furthermore, these incompatibilities can be spread across different machines, leading to cases where a choice on the first

machine in a scenario would lead to problems when choosing a procedure for a transition on the last machine, causing every decision between the two to be reversed and increasing computation times significantly. Each time a transition dealing with secrets is encountered, we check if that secret has been encountered before in the refinement process. If not, the procedure gives this secret a type and a value according to its secret preconditions $\pi_{secrets}$. If the secret has been handled before, this means that it has been given a value and a type by another selected procedure. In this case, only transitions with compatible secret preconditions are eligible for refinement.

Formally, we define this backtracking refinement process in algorithm 1.

Note that any refined scenario (described on a procedural level) will be executable/winnable as long as a) the scenario described on a technical level was itself executable/winnable, b) a valid combination of procedures is found, c) procedure constraints are properly implemented. In fact, any winning attack path available to the attacker on a technical level can also be followed on a procedural level by chaining the procedures corresponding to the techniques executed in that attack path. The complexity of this algorithm depends heavily on the number of procedures available for each technique in the scenario, as well as how hard (i.e., how much backtracking is required) it is to refine said scenario. Due to the recursive nature of the algorithm, a strict upper bound on complexity would be $O(t * p)$, where t is the number of transitions within the scenario and p is the maximum number of procedures associated with each technique. Such a bound would be reached if every procedure choice resulted in the very last transition of the scenario being impossible to choose a procedure for. In practice, time complexity problems were never encountered during the deployment process, in part due to our decision to prioritize transitions dealing with secrets. One might argue that this is also due to the size of our scenarios, but in any case, deploying the scenarios took orders of magnitude longer than generating them.

In some cases, such as when the resulting version of a software or OS is "*" or when an OS type is simply set to "linux", there is a need to pick a specific version or type. For software, this usually ends up being the most easily available version of that software. This choice is handled by the virtual machine deployment part of URSID, which we detail in the next section. For operating systems, we have a set list of OS type - consisting for now of Windows, Ubuntu and Debian distributions - and their versions. In case more than one of those OS is applicable to the constraint by the end of the refinement process - for instance `{"Type": "canonical:ubuntu_linux", "Version": "*"}`, we take one at random within all the OS that fit this particular constraint.

Figure 3.6 showcases the result of that procedural refinement as part of URSID’s full workflow. Having been converted into sets of constraints, procedures are checked against each other for incompatibilities. Such incompatibilities arising lead to impossible architectures to deploy, and are thus avoided thanks to the backtracking refinement algorithm. In the event where a valid combination of procedures is found, the resulting architecture is coherent and will thus be able to be deployed.

3.2.3 Example: APT-3 inspired scenario.

We will now showcase this algorithm using the same APT-3 based example as in chapter 2. As a reminder, the technical level representation of this scenario is available in figure 3.7.

Each of the techniques available in this scenario have been associated with one or more procedures, all with varying types of constraints. We offer a summary of all those procedures and techniques in table 3.8.

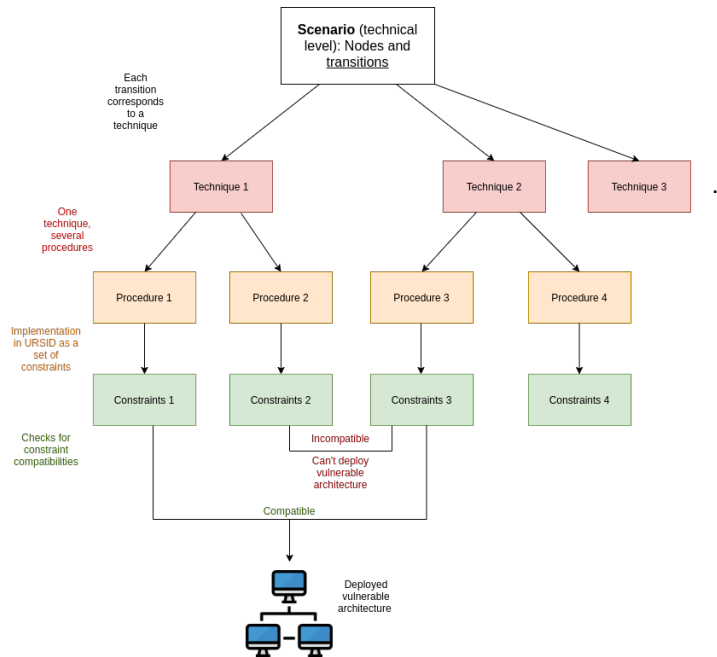


Figure 3.6 – URSID full workflow.

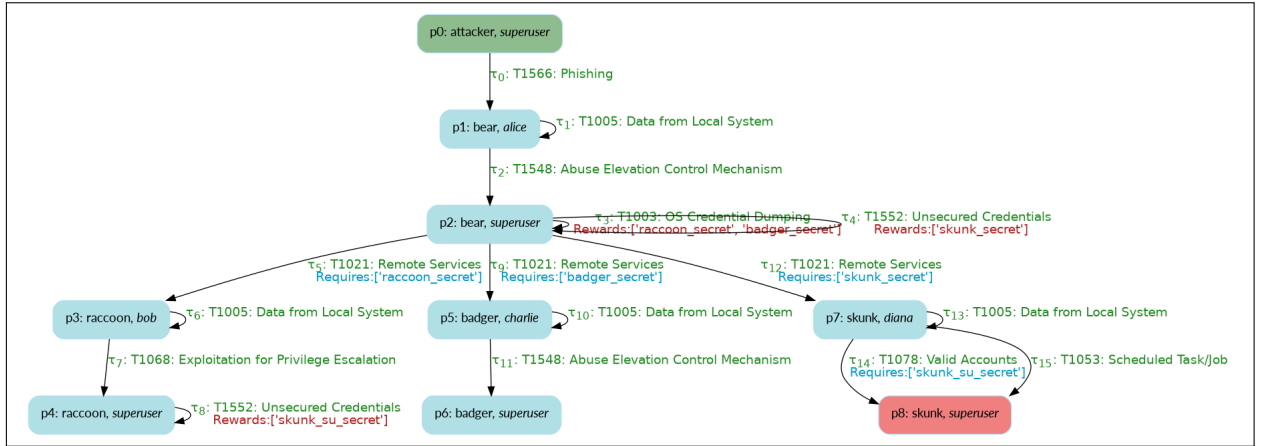


Figure 3.7 – Formalized example scenario based on APT-3 a technical level.

Algorithm 1 Backtracking refinement algorithm for a scenario $\mathbf{S}^{\text{tech}} = (\mathbf{P}, \mathbf{A}^{\text{tech}})$ ⁶

Require: $\mathbf{S}^{\text{tech}} = (\mathbf{P}, \mathbf{A}^{\text{tech}})$ a scenario described at the technical level ($\mathbf{A}^{\text{tech}}$ are attack techniques)

Require: $\mathbf{C}$ a set of architectural constraints.

Require: $\{\pi_{\tau_i}\}_i$ a list of procedures instantiating each τ_i

$\mathbf{S}^{\text{proc}} := (\mathbf{P}, \emptyset)$

$L_t := \mathbf{A}^{\text{tech}}$ the list of all transitions to refine, sorted to prioritize transitions requiring or rewarding secrets.

Ensure: $\mathbf{S}^{\text{proc}} = (\mathbf{P}, \mathbf{A}^{\text{proc}})$ a scenario described at the procedural level ($\mathbf{A}^{\text{proc}}$ are attack procedures) and $\mathbf{S}^{\text{proc}} \triangleright \mathbf{S}^{\text{tech}}$

function BACKTRACK($\mathbf{S}^{\text{tech}}, \mathbf{S}^{\text{proc}}, L_t, \{\pi_{\tau_i}\}_i, \mathbf{C}$)

if $L_t = \emptyset$ **then return** True, $\mathbf{C}$

else

 Sort L_t , putting transitions requiring or rewarding secrets first.

$t = L_t[0] = ((u1, m1), (u2, m2), (\tau, \mathbf{pre}, \mathbf{post}))$

if $\{\pi_{\tau}\} = \emptyset$ **then return** False, $\emptyset$

end if

 Get the set of all compatible procedures $\{Comp(t)\}_i$

for all $\{\pi \in Comp(t)\}$ **do**

$\mathbf{C}_{new} = \mathbf{C}$

$\mathbf{C}_{new}[m1] = \mathbf{C}_{new}[m1] \wedge \pi[m1]$

$\mathbf{C}_{new}[m2] = \mathbf{C}_{new}[m2] \wedge \pi[m2]$

if $\mathbf{C}_{new} \not\models \perp$ **then** #If the new constraints are not inconsistent

$\mathbf{S}_{new}^{\text{proc}} = (\mathbf{P}, \mathbf{A}^{\text{proc}} + \pi)$

$L_{tnew} = L_t - \{t\}$

 isValid, $\mathbf{C}_{final} = \text{backtrack}(\mathbf{S}^{\text{tech}}, \mathbf{S}_{new}^{\text{proc}}, L_{tnew}, \{\pi_{\tau_i}\}_i, \mathbf{C}_{new})$

if isValid **then return** True, $\mathbf{C}_{final}$

end if

end if

end for

return False, $\mathbf{C}$ #No valid refinement was found

end if

end function

Technique	Possible procedures	Constraints and secrets summary for each procedure
$\tau_0 =$ T1566	π_1 : Linux Payload π_2 : Windows Payload	Linux Windows
$\tau_{1,6,10,13} =$ T1005	π_3 : Business File Corpus π_4 : Student File Corpus	Adds Files Adds Files
$\tau_2 =$ T1548	π_5 : Bad Sudo Config π_6 : CVE to disable UAC	Linux, edits sudoers Windows, installs flawed software
$\tau_3 =$ T1003	π_7 : Crackable Password in /etc/shadow π_8 : SAM dump	Linux, a crackable hash, rewards a plaintext password Requires Windows, rewards a hashed password
$\tau_{4,8} =$ T1552	π_9 : Password in bash history π_{10} : Private keys in .ssh π_{11} : Passwords in text file	Linux, edit bash history file, rewards a plaintext password Requires Linux, creates files using a secret, rewards SSH keys Creates a file using a secret, rewards a plaintext password
$\tau_{5,9,12} =$ T1021	π_{12} : Weak SSH password π_{13} : SSH access from key π_{14} : SSH access from password π_{15} : RDP from Password	Linux, ssh service on port 22, sshd_config edit Linux, ssh service, authorized_keys edit Linux, ssh service, ssh_config edit Windows, RDP service, Windows registry + user password edit
$\tau_7 =$ T1068	π_{16} : Dirty COW π_{17} : Vulnerable sudo version	Specific Linux version Requires Linux, specific sudo package, edit sudoers
$\tau_{15} =$ T1053	π_{18} : Cronjob with bad permissions π_{19} : Scheduled Job with bad permissions	Linux, files edits Windows, files edits
$\tau_{14} =$ T1078	π_{20} : Password from secret π_{21} : Weak Password	user password matches a secret user password easy to brute-force.

Figure 3.8 – Available procedures for each technique in the APT-3 inspired scenario.

The scenario was then refined using our backtracking refinement algorithm. We ran this process several time in order to check the range of results. Remarks on the process are as follows:

- The algorithm initially frequently backtracked, due to the fact that τ_{14} is one of the last transitions to be refined, and deals with a secret defined earlier in τ_8 . Based on the procedures implemented, τ_8 can reward either a SSH key or a password, while τ_{14} only accepts passwords. Therefore, picking a SSH key related procedure for τ_8 would then cause multiple backtracking on every procedures taken after. This is why we now handle procedures which deal with secrets first, which reduced computation times significantly.
- Having a procedure decide whether the machine runs on Windows or Linux quickly limits the options for the next procedures on the same machine.
- The choice of procedure for technique T1552: *Unsecured Credentials* has consequences on the techniques available for T1021: *Remote Services* between *(bear, superuser)* and *(skunk, siana)*. Indeed if T1552 rewards a SSH key to the attacker, this instance of T1021 cannot pick a procedure requiring a password (for instance a RDP connection).
- The procedure chosen for technique T1021: *Remote Services* between *(bear, superuser)* and *(raccoon, bob)* can only be a procedure fit to require a secret, and thus cannot be "Weak SSH Password".
- Since T1078: *Valid Accounts* procedures only accept passwords, T1552: *Unsecured Credentials* between *(Raccoon, SuperUser)* and itself may not give a SSH key as a reward.
- The output architectures feature varying numbers of Windows and Linux operating systems. Machine *raccoon* must be on a Linux (because of our available procedures for T1068), but *skunk*, *badger* and *bear* can be either.
- The number of possible resulting architectures exponentially grows as we increase the number of machines deployed, procedures implemented or amount of OS we are able to deploy. However, the refinement backtracking algorithm only picks out one of those results, making the actual computation complexity manageable⁷.

Figure 3.9 showcases an example of a refined scenario obtained using this algorithm. As one can see, each procedure chosen does correspond to the technique that was associated

7. A few seconds at most for this example.

with that transition. This specific example is interesting as it only contains Linux procedures, which will make it easier to deploy later. A snippet of the resulting configuration file (expressed in the json format) is available in figure 3.10. This snippet showcases the configuration of machine *raccoon* in this particular choice of procedures. This machine runs on the Ubuntu operating system at version 20.04. This choice was made at random between every Linux distribution available in URSID, as the procedures involved with this machine only required a Linux system. This machine requires the SSH software to be installed - which will be ran on port 22 - in order to exploit τ_5 : *Remote Services, SSH Access from password*. The sudo package is also installed at a specific version in order to perform τ_7 : *Exploitation for Privilege Escalation: Vulnerable sudo version*. Both users on the machine have randomly generated passwords, although Bob's password is set by τ_5 and will match the one found earlier in the scenario.

Finally, files related to the password leaking made possible by τ_8 : *Unsecured Credentials, Passwords in text file*, and the corpus of files made by τ_6 : *Data From Local System, Business File Corpus* are added to the machine. Note that the **Credentials** field of each of the users has a **Type** field, which is equal to **UnixUserAccount** in both cases. This is there to represent that the credential has to be used for the account of the user on the operating system. While this is so far the only type of account credential types implemented in URSID, this was added for future proofing reasons as we hope to add other types (such as Windows Active Directory accounts) in the future.

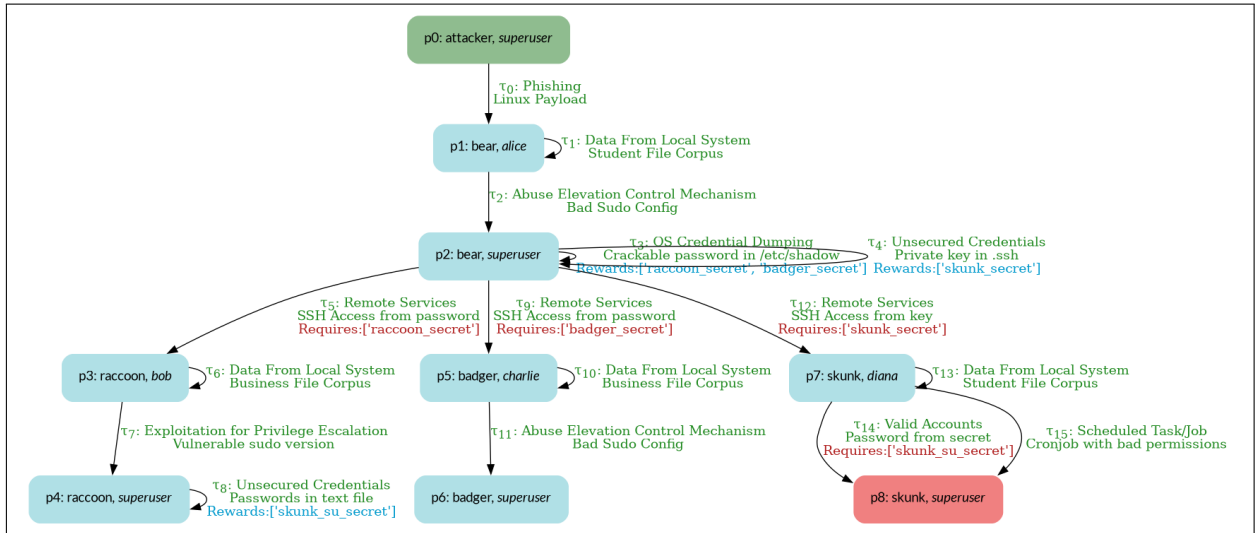


Figure 3.9 – Possible refinement result for APT-3 inspired scenario.

```
1 [{ "Name": "Raccoon",
2   "OS": {"Type": "canonical:ubuntu_linux", "Version": "20.04"},
3   "Software": [ {"Type": "sudo_project:sudo", "Version":
4     ↪ "1.8.27"},
5     {"Type": "ssh", "Version": "*", "Port": "22"}],
6   "Users": [
7     { "Name": "bob",
8       "Group": "bob",
9       "Privilege": "User",
10      "Services": ["ssh"],
11      "Credentials": { "Type": "UnixUserAccount", "Value":
12        ↪ "n80KZ7Yr1ohxYfF0vOegXQ"}},
13     { "Name": "superuser", "Group": "superuser",
14       "Privilege": "SuperUser",
15       "Services": [],
16       "Credentials": { "Type": "UnixUserAccount",
17         "Value": "cHL2l5I7tEV2BhGKZLXaNQ"}},
18   "Files": [
19     { "Destination": "~/important.txt",
20       "Source": "./generated_files/raccoon/important.txt",
21       "Modification": "WRITE",
22       "Owner": "superuser",
23       "Group": "superuser",
24       "Permissions": "0700"},
25     { "Destination": "~/business",
26       "Source": "./generated_files/raccoon/business",
27       "Modification": "WRITE",
28       "Owner": "bob",
29       "Group": "bob",
30       "Permissions": "0700"}]
31 ]
32 ]
```

Figure 3.10 – Snippet of the configuration file for the scenario resulting from the procedural refinement showcased in figure 3.9.

3.3 Conclusion

In this chapter, we introduced our solution for translating technical-level scenarios to a procedural level through a process we call procedural refinement. By tying machine configuration to procedures through architectural constraints, we are able to both check for

inconsistencies as we select procedures and have enough information about each machine at the end of the process to deploy the scenario. In this way, we are able to describe and deploy scenarios at a high level, providing a solution to the problems we have identified in the state of the art. While there is still some manual work required to perform this refinement process-in particular, writing out procedures and their architectural constraints-this work only needs to be done once. Now that we have selected policies and have information about the configuration of each machine within the scenario, we still need to convert this information into configuration files for deploying software to virtual machines. The next chapter of this thesis will discuss how this conversion is done and how the APT-3-inspired scenario was deployed and played.

DEPLOYMENT OF VULNERABLE ARCHITECTURES.

The previous chapter showed how URSID was able to refine a scenario described on a technical level into one or more instances of that scenario defined on a procedural level. This process is achieved by selecting a procedure for each transition of the scenario, which is defined by its set of architectural constraints. Once this process is complete, i.e. a suitable combination of procedures has been found for the scenario, a configuration file is generated containing all relevant information about the resulting architecture. This includes the operating system, software, accounts, and files to be installed on each machine. However, this configuration file cannot be passed directly to virtual machine deployment software. This chapter of the paper will explain this conversion process before presenting an example deployment using the APT-3-inspired scenario mentioned earlier.

4.1 Goal overview.

The refinement process outputs a configuration file (written in JSON), containing information about every machine within the architecture. In particular, this consists of:

- The name of the machine.
- The type and version of operating system that must be installed on the machine.
- The list of type and version of software that must be installed.
- The list of account names, groups, privileges, services and credentials that must be installed.
- The list of file local paths, destination paths, modifications, owner, group and permissions that must be added. Files content is already generated (and available at the local path).

- (Optional) the list of networks the machine takes part in. If not present, every machine is assumed to be part of the same network.

These parameters were chosen because they encompass everything needed to put the machine in the desired state to play out the scenario. Furthermore, they are naturally derived from our choice of architectural constraints¹. However, this information still needs to be parsed before it is passed to a virtual architecture deployment software. In addition, a deployment option must be selected in the first place.

We organized and supervised a student project aimed at providing a first technical solution to these problems. They were tasked with evaluating virtual machine deployment options and helping to design the parsing layer that converts URSID output into configuration files suitable for the chosen virtual machine deployment solution. We met regularly with the students to monitor their progress and discuss the best ways to combine their own engineering work with the URSID refinement output at that time. The goals of their project were defined as follows:

- Test and compare virtual machine deployment technologies and figure out which one suited our needs best.
- Deploy a test instance defined by URSID, which contained at least 2 machines and a service.

Their work over this 6-month period part-time was overall very satisfactory and is still used as a basis for the deployment aspect of URSID to this day, which they dubbed *Apiculteur*².

For technologies, the choice was made on a combination of VirtualBox, Vagrant and Ansible:

- VirtualBox Oracle, *op. cit.* is a free and Open Source virtualization software developed by Oracle. It was chosen over other virtualization options such as VMWare for being free of use. Other free virtualization solutions do exist (such as KVM), but VirtualBox was determined to be the easiest and fastest solution to get a working deployment with, while being innately compatible with other software such as Vagrant.
- Vagrant is a virtual machine management tool, which can communicate with virtualization software (such as VirtualBox) in order to deploy and manage several virtual instances at once. It was chosen over Terraform for being a more lightweight option:

1. with the exception of networks, which come from our optional configuration file.

2. French word for beekeeper.

Terraform is more suited for massive scale cloud operations, while our goal at the time was to deploy local instances. Docker containers were also considered, but were judged to be less close to an actual physical machine as a virtual machine would be. Credibility of the deployed instance was of an even greater concern at the time, as this project started as a honeypot - vulnerable infrastructure designed to lure attackers in - deployment tool.

- Ansible is a virtual machine provisioning tool, which can communicate with deployed instances in order to perform actions on them, known as tasks. This may for instance consist of installing software on the machine, running commands or copying files from the host. Ansible is the most commonly used provisioning software to go with Vagrant and seemed able to perform every task we wanted, making it a straightforward choice.

We thus needed to convert the JSON files issued from URSID's refinement process into configuration files that fit this VirtualBox/Vagrant/Ansible infrastructure. Vagrant makes use of Vagrantfile in order to specify information about the virtual machines to deploy. Written in Ruby, this file is able to specify the name and operating system of the virtual machines, assign them IP and networks, then signal Ansible to start provisioning these machines. Ansible uses playbooks, written in YAML. These playbooks contain a list of tasks which are to be performed in order on the machine. Each playbook by default is associated to a single machine. Finally, no specific configuration for VirtualBox is required. Parts of the deployment - such as the ones related to networking - may however still require communicating with VirtualBox, in which case we make use of the `vboxmanage` utility.

In the event where more than one instance of a scenario was generated (which is one of URSID's main perks), machines in each instance are numbered in order to differentiate them. For instance, if a scenario has a single machine *machine*, and we choose to deploy two instances of it (by operating the refinement process twice), the resulting machines will be named *machine0* and *machine1* respectively. Deployment files for each instance (Ansible playbooks, Vagrantfiles, files to copy...) will also be in their own directory numbered similarly.

Apicuteur first parses the URSID JSON configuration file in order to acquire information about each machine. It then converts this parsing result in a Vagrantfile and Ansible playbooks. We will now detail how this conversion process is performed.

4.2 From URSID to Vagrant/Ansible.

Output structure The first version of URSID had a single Ansible playbook per machine. These playbooks could all be run at once or separately in order to deploy the scenario. Due to the increasing scope of our deployment plans however, we now make use of Ansible roles Ansible, *Roles*, URL: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_reuse_roles.html (visited on 05/30/2024). Ansible roles are a structured way to organize tasks and task-related information for a given machine. This role can then be deployed by itself as part of another playbook. We make use of the following role options during URSID deployment, which are all defined in their own directory:

- *tasks*, which contains the instructions necessary to provision this particular machine.
- *files*, which contains files which will have to be copied to the machine.
- *vars*, which defines variables (names and values) which can then be used in other tasks. These variables are generated by URSID itself at first then added to this directory. We will mention throughout this section whenever we create or make use of variables.
- *templates*, which contains files whose contents may have to be dynamically edited using variables.

A playbook is still generated for each machine separately for implementation reasons, which only contain the code necessary to run their related role. A main playbook is then generated which combines the code of all these playbooks.

A Vagrantfile is also generated for each scenario. This file contains information needed by Vagrant in order to create and boot up the virtual machines, such as OS or networking configurations . We will mention whenever we make use of the Vagrantfile in the later sections. In the event where another virtual machine deployment tool is used - such as Terraform - this Vagrantfile generation is made optional, although the user then has to provide their own solution.

Figure 4.1 shows the resulting directory structure after the refinement of a scenario into two instances. `output_scenario.json` is the configuration file resulting from the refinement process.

Networking The current implementation provided with URSID makes use of VirtualBox networking abilities. By default, VirtualBox machines are given IPs belonging to the private

```
- scenario_0
  - roles
    - machine0
      - files
      - tasks
      - templates
      - vars
    - main_playbook_machine0.yml
    - output_scenario.json
    - Vagrantfile
- scenario_1
  - roles
    - machine1
      - files
      - tasks
      - templates
      - vars
    - main_playbook_machine1.yml
    - output_scenario.json
    - Vagrantfile
```

Figure 4.1 – Directory structure for two instances of a scenario.

network range 192.168.0.0. We first ask VirtualBox which subnetwork ip are available, which may depend on whether virtual machines are already deployed. In our current implementation, the first subnetwork (if available) will be created with the address range 192.168.56.0, with subsequent networks being 192.168.57.0, 192.168.58.0, etc. This is both useful for scenarios which contain more than one subnetwork, and for when we wish to deploy more than a single instance of a scenario. The IPs and subnetwork configurations are thus chosen by VirtualBox and then provided to Vagrant in order to deploy the machines according to those patterns. These subnetworks are set up in the Host-Only Adapter networking mode Oracle, *Introduction to Networking Modes*, URL: <https://www.virtualbox.org/manual/ch06.html> (visited on 05/30/2024), making it possible for the boxes to communicate with the host and with each other. Being able to communicate with the host is necessary for the provisioning process. Each machine is also equipped with a NAT adapter to communicate with the Internet. This is particularly important for the provisioning process to download packages or install software, but comes at the cost of network credibility, as machines that should not be able to talk to each other may still be able to do so. In practice, while this may theoretically allow attackers to access

machines and attack positions at earlier points in the scenario than expected, any transition requiring secrets-which often leads to new machines and positions-will still require the attacker to follow the designed attack path, thus significantly reducing any sequence break made possible by this simplification. Nonetheless, this weakness of URSID's native networking abilities will be kept in mind in future deployment during live experiments, which will circumvent this issue by using more advanced network configurations allowing for better machine encapsulation.

The IP of each machine is added as a variable to the var folder under the name `IP_machinename`. This is useful for specific use cases where a machine needs to know the structure of the network, such as when setting up DNS servers. If more than one IP is given to the machine (for example, if it is part of two different subnets), one of them will be chosen at random due to implementation limitations. We acknowledge that this could lead to complications in the future - and is one of the reasons that the main focus of improvement for the URSID implementation is on its networking capabilities - but this has been sufficient for our applications so far.

This VirtualBox-based networking implementation is the default mode of URSID and is particularly efficient for deploying instances locally on a host. However, it became apparent that more ambitious implementations could result in the selection of IPs not being handled by VirtualBox. This was the case in both experiments conducted with URSID throughout this thesis, as both required the architecture to be available remotely rather than deployed locally. To account for these cases, it is possible to disable the Vagrant and VirtualBox networking part of URSID along with its Vagrant implications. However, the user must then provide their own deployment solution and IP ranges. Networking variables are still added for each machine. We use a combination of our optional YAML configuration files (which can specify IP ranges of machines) and bash scripts to fill these entries.

Operating Systems The configuration file resulting from the refinement process provides us with the name (under our summarized CPE format) of the operating system which has to be implemented on each machine. These names are converted into the name of publicly available Vagrant boxes which fit the given OS. For instance, `debian: 9.0` is converted into `debian/stretch64`. This name can then be directly provided to Vagrant (via the `Vagrantfile`) which will handle the downloading process. While we would ideally provide our own boxes in order to guarantee their contents, in practice we found these publicly available boxes to be fairly consistent. These virtual machine images are configured the

bare minimum, for instance missing basic software such as the gcc compiler.

Software Similarly to operating systems, software information is provided with its name and version. As we are only able to deploy Linux instances so far and chose to focus on Debian distributions, these software are usually linked to .deb packages. In most cases, these packages are directly available with a simple apt-get command, which is natively handled by Ansible. However, some software requires special installation instructions. This may happen for software which are not available immediately in online repositories , requiring the addition of a repository key. Nodejs, a JavaScript runtime build used for web applications was one such instance. In other cases, the software has to be compiled manually or requires specific operating system actions beforehand. For instance, installing a chosen version of the sudo package, used to handle permissions within a Linux system, first requires uninstalling the existing sudo version, before downloading and manually compiling a new one. In order to solve this issue, URSID has a list of these special cases and provides generic bash scripts which handle the installation process. These scripts will be copied to the /tmp directory of the deployed machine, executed and then deleted.

Some software also requires other software to be installed in order to be properly added to a machine. For example, software that needs to be compiled (such as sudo) may require the make or gcc packages. Fortunately, URSID's procedure constraint system makes it possible to associate these additional required installations with the procedure itself. For any software installed on a machine, the version of that software is added as a variable to the Ansible role. This is useful for the aforementioned installation scripts, which are written as templates to be as generic as possible.

Unfortunately, despite our best efforts at genericity, some software simply behaves differently depending on the operating system. For example, some packages are simply not available on old operating systems like Debian 7.0 or Ubuntu 14.04. We've dealt with this problem by simply excluding these operating systems from the list of ones we can deploy, but a more complete implementation would require considerable additional engineering work, writing installation scripts that detect which platform they're on and change their installation instructions based on the result. Note, however, that the URSID constraint system is again very useful for deployment: if a given procedure requires software that is difficult or impossible to install on old operating systems, these OS conditions can be specified within the procedure itself.

Figure 4.2 showcases a few Ansible tasks related to software. First, the postgresql

```
- name: Install package postgresql
  become: yes
  apt:
    name: postgresql
    allow_unauthenticated : yes
    update_cache: yes
    update_cache_retries: 10
    update_cache_retry_max_delay: 300

- name: Copy install file netcat at version '*'
  become: yes
  ansible.builtin.template:
    src: install_netcat.sh
    dest: /tmp/install_netcat.sh
    owner: root
    group: root
    mode: 0755

- name: Install netcat at version '*'
  become: true
  shell:
    /tmp/install_netcat.sh
```

Figure 4.2 – Software related Ansible tasks.

package is installed, which does not require any specific configuration and can be fetched directly using the apt package tool. The `update_cache` options are here to solve some niche timeout cases. Then the netcat package is installed. This one requires a specific installation script, as the package to install is named differently on Ubuntu and Debian, with the names `netcat` and `netcat-traditional` respectively. The `install_netcat.sh` file is a bash installation script which takes care of both cases. This file is first copied to the `/tmp/` folder with appropriate permissions then executed.

Accounts Ansible natively handles account creation as long as the username is provided, which we are able to fetch from the post-refinement configuration file. This file also provides us with credential information about that account, which Ansible can set after we hash and salt it properly. As a reminder, Account constraints had two possible levels of privileges: User and SuperUser, an information which is still available in the configuration file. In the even where this value is at SuperUser, we create a task which adds the name of this user to the `/etc/sudoers` file. This file is available by default on most Linux systems file, and

is used to allocate system rights to users. By adding an entry to this file, we can make sure our account with SuperUser privileges can use the `sudo` prefix to any command, thus giving them administrative rights on this machine.

Accounts may also be associated to services, which were thus far represented only by their name (such as "npm" or "django"). Services in Linux are managed by the `systemd` software suite³, which in turn can be managed by the `systemctl` command-line tool. Services may be added by adding a file to the `/etc/systemd/system` directory of the machine. This file specifies information such as the owner and group of the service (which here both correspond to the name of the account) as well as the path of a file which has to be executed by the service at launch. We make this path point to a bash script we store in `/usr/bin`, which contains the code necessary to run the service. One particular service we often used was launching a Docker instance. Docker is a deployment service which is able to deploy applications inside virtualized lightweight packages known as containers. It was particularly used during collaborative scenario deployment using URSID in which other researchers developed their own services (such as websites) as standalone then packaged it using Docker for collaborative use, making the development process easier on their end. Docker containers are however harder to debug and longer to install than launching the application directly. Some procedures thus indicate (thanks to their account constraints) that they require a Docker container to be launched as a service, which URSID is able to deploy.

Whenever an account is added to a machine, a home directory for that user is also created. We made the choice of changing the default permissions of that account to 750, which translates to no access to that directory for other users on the machine, excepted ones with administrative privileges⁴. This was done in order to limit the path of the attacker to positions they control, as without this an attacker would be able to fetch files from users they are not able to log as. This restriction also applies to any possible default account available on a machine (such as Ubuntu, Debian or vagrant), which are used to communicate with it during the deployment process. The password to the vagrant user is also changed, as by default this user has a) administrative privileges on the machine b) a very simple password, which would make it trivially easy to perform privilege escalation on the machine should the attacker notice this user. Finally, some services (such as ssh) may already exist by default on the machine and thus do not require any further installation.

3. At least in every distribution we used in URSID.

4. And users within the same group, but this is most of the time only one user.

In these cases, we consider that a user requesting these services means they only require a restart of those services, which is useful for configuration changes (such as making it possible to log through SSH via passwords). We consider that a service already exists on the machine if no URSID installation file is found for it.

Figure 4.3 offers a few examples of account related tasks. The first half of these tasks creates a user *bob* and changes the permissions of its home directory to 750. The second half creates a service (here related to the python web framework Django). This is done by creating an executable file at `/usr/bin/django_service.sh` and a service file in the `/etc/systemd/system/` directory, before launching this service and making sure it will be automatically launched at any boot.

Files Files contents were generated during the refinement process itself. These files were either copied to the *files* directory of the machine's role or the *templates* directory, depending on whether they interact with variables (defined in the *vars* directory) or not. Whenever a variable is used in a template file it is indicated as `{{variable_name}}`. As a reminder, variables include information such as the IP of the machine or owner of services. Note that

At the end of the refinement process, information about files was as follows:

- A local path, corresponding to its name of the host machine.
- The path on the deployed machine.
- The owner and group of the file on the deployed machine.
- Permissions of the file on the deployed machine (written in octal Linux format).
- The action that needs to be performed on the file, which can be one of the following:
 - **WRITE**: The file must be directly copied from the host machine to the deployed one.
 - **WRITE TEMPLATE**: The file must be copied as an Ansible template, which will fetch appropriate variables in *vars*.
 - **APPEND**: The content of the file must be extracted and added to the end of an existing file on the deployed machine.
 - **APPEND BEGINNING TEMPLATE**: The content of the file must be extracted and added to the beginning of an existing file on the deployed machine. The content of the file contains variables which have to be edited using Ansible templates.

```
- name: Add user bob
  become: yes
  ansible.builtin.user:
    name: "bob"
    password: "$6$GcXFSEBr$gPYuyCuq3ZgGDsE9/liZIykAoGvjckZ09[...]"
    shell: "/bin/bash"
    create_home: yes

- name: Check whether bob directory exists.
  become: yes
  ansible.builtin.stat:
    path: /home/bob
  register: directory_exists
- name: Change bob permissions to 750 if exists.
  become: yes
  ansible.builtin.file:
    path: /home/bob
    mode: 0750
  when: directory_exists.stat.exists

- name: Adding script to execute django_service.sh to /usr/bin
  become: yes
  ansible.builtin.template:
    src: django_service.sh
    dest: /usr/bin/django_service.sh
    owner: bob
    group: bob
    mode: 0755

- name: Adding service file to /etc/systemd/system to be executed as user bob
  become: yes
  ansible.builtin.template:
    src: django_service.service
    dest: /etc/systemd/system/django_service.service
    owner: root
    group: root
    mode: 0644

- name: Launch service and enable it at boot
  become: yes
  ansible.builtin.service:
    name: django_service
    state: restarted
    enabled: yes
```

Figure 4.3 – Account related Ansible tasks.

- **EDIT {line_content}**: Looks for a line containing `{line_content}` within the file on the deployed machine, and replaces it with the content of the file on the host.
- **EXECUTE**: Executes the contents of the file as a bash script. Is assumed to be a template.

URSID reads this information and converts it into the appropriate Ansible task, using a combination of the `file`, `copy`, `template` and `lineinfile` builtin Ansible modules and functions. Note that the **WRITE** operation may also be applied to a directory. In this case, instead of relying on the default Ansible copy function, we made the choice of synchronizing the directory on the deployed machine with the one on the host using the `rsync` package. We then edit the permission of this directory to match the one indicated by the configuration file, as the synchronization will first give them the same permissions they have on the host. This is done because the Ansible copy function is very slow to use when applied to a large number of files, regardless of their size. This is also why we install the `rsync` package on every machine by default (which we indicated by using software constraints during the refinement process).

Figure 4.4 showcases a few Ansible tasks related to files. The first two copy a file named `add_ssh_public_key` on the host machine to the `.ssh` directory of the user *superuser* on the deployed machine. The next three tasks copy a directory named `django_website` on the host to the home of user *bob* on the deployed machine, by using Ansible `synchronize` module (which makes use of the `rsync` package we mentioned earlier).

Post tasks Some tasks need to be executed after others in order to be functional. For instance, tasks related to restarting a service need for that service to be installed in the first place. We solve some of these issues thanks to the order in which we choose to write tasks. For instance, we perform software related tasks before files tasks, as these tasks may try to access configuration files related to the installation of these software. In some cases however this is not sufficient, as some tasks need to be performed at the very end of the deployment process. For instance, tasks related to restarting a service need for that service to be installed in the first place. We solve this issue by writing these tasks as post tasks instead, which are natively executed after every task is complete by Ansible. At the time of writing these tasks include:

- Changing the password of the *vagrant* user if it exists, in order to avoid attackers making use of default credentials for easy privilege escalation.

```
- name: Make sure /home/superuser/.ssh directory exists
  become: yes
  file:
    path: "/home/superuser/.ssh"
    owner: "superuser"
    group: "superuser"
    state: directory

- name: Copy add_ssh_public_key file to /home/superuser/.ssh/authorized_keys
  become: yes
  ansible.builtin.copy:
    src: "add_ssh_public_key"
    dest: "/home/superuser/.ssh/authorized_keys"
    owner: superuser
    group: superuser
    mode: 0644

- name: Make sure /home/bob directory exists
  become: yes
  file:
    path: "/home/bob"
    owner: "bob"
    group: "bob"
    state: directory

- name: Copy django_website directory to /home/bob/
  become: yes
  ansible.posix.synchronize:
    dest: "/home/bob/"
    src: "django_website"
    use_ssh_args: true

- name: Change django_website permissions
  become: yes
  ansible.builtin.file:
    recurse: yes
    path: "/home/bob/"
    owner: "bob"
    group: "bob"
    mode: "0755"
```

Figure 4.4 – Files related Ansible tasks.

- Clear the `/tmp` directory of the machine which was used for all installation scripts, in order to avoid giving information about the machine’s software configuration to attackers.
- Service restart related tasks.

Discussion The networking part is by far the less mature part of URSID. While its current reliance on VirtualBox to acquire IPs and create a network is adequate for local use, both of the experiments we ran with it had their own specific networking setup required for remote access to the instance. It is still manageable to customize the current URSID output for these specific use cases, but the extra work required is something we are looking to improve in the future. It is also fairly plug-and-play in its current state if you want to deploy an architecture on a local network. Our choices of implementations in order to convert OS, Software, Account and Files configurations into Ansible tasks cover a wide range of use cases we encountered. New tasks can (and were) added in order to deal with edge cases, such as specific software installation which require more than a single bash script to be executed.

We combined the technical level scenario and refinement process code - which we dubbed Chouchen⁵ - with Apiculteur into a single repository. This repository is licensed under the open source GNU AGPL v3 license, and is publicly available online BESSON, *op. cit.*, alongside deployment instructions and detailed documentation Pierre-Victor BESSON, *URSID documentation*, URL: <https://ursid.readthedocs.io/> (visited on 05/30/2024).

The resulting workflow in order to create and deploy a scenario from scratch, as of the time of writing this thesis, is therefore as follows:

- First, the user formalizes the desired scenario on a technical level. This can be done either by using URSID’s native GUI (recommended) or by writing code. This description includes the attack positions of the scenario, any transitions between these positions, as well as the secrets required or rewarded by these transitions. The list of available techniques corresponds to those that have procedures implemented within the URSID procedure library. Thus, if a technique desired for the scenario is missing, one must implement new procedures for that technique. This requires writing its OS, software, file and account constraints, but may also require adding new components to Apiculteur, the part of the software responsible for converting the refinement output into virtual machine configuration files. This can happen if

5. Named after a Briton honey-based alcohol, also the former name for this thesis.

the constraints associated with the procedure are difficult to install, such as specific software whose installation or configuration is not directly available through tools like the Linux APT utility.

- If desired, the user may force some transitions to use specific procedures. If done so, these procedures will be taken as granted for the refinement process, and the corresponding transition will not be considered for backtracking.
- The user specifies how many variations of the scenario should be generated, then performs the refinement. This part does not require any input from the user. The refinement outputs a list of architectural constraints corresponding to a valid list of procedures which fit the original scenario. For each variation of the scenario asked by the user, an additional backtracking and resulting output of constraints is made available. This is where the work of the Chouchen part of the software ends.
- Each of these lists of constraints are then inputted into Apiculteur, which converts them into Ansible playbooks (used to provision the machines) and Vagrantfiles (used to deploy the virtual machines on a local network).
- In the event in which the user wishes to deploy the machines locally, the machines may be directly deployed through the use of the `vagrant up` command. If a custom networking architecture is desired (for instance, uploading the instances on a remote cloud network), the user may discard the Vagrantfiles in favor of their own implementation.

This workflow was the one used throughout this paper, as well as during the two experiments conducted using URSID, which we will discuss in a later section. Note that this software is still under active development by members of the research team, and numerous improvements were made to it late in the writing of this manuscript to make it easier to add new procedures and use a customized deployment setup. We will focus on the state of the software as it was available during the work on this thesis, but the online repository of URSID—which is open source as part of the contributions to this thesis—is still being updated.

APT-3 inspired scenario deployment The APT-3 inspired scenario, which we have been using as an example throughout this thesis, was the first scenario to be refined and deployed using URSID.

The specific instance - corresponding to a choice of procedures - we deployed is showcased in figure 4.5. Table 4.1 offers a reminder of the procedures themselves, how they

were implemented and how the attacker is expected to perform them. A few remarks of the development and deployment of this specific instance are as follows:

- The payload entry point - associated with τ_0 : *Phishing: Linux Payload* - was simulated with a bash script on the machine which could be remotely activated through SSH. This could also have been accomplished with a cron job (Linux way of handling scheduled tasks), but we needed to be able to control the timing of the detonation for demonstration reasons.
- Installing specific versions of the sudo package - required for τ_7 : *Exploitation for Privilege Escalation: Vulnerable sudo version* - is nontrivial: the current version of sudo has to be uninstalled (which may cause issues), and the new version has to be downloaded and compiled manually. This is what lead to our current implementation for software, in which specific edge cases may be installed with the use of templated bash scripts instead. The way sudo handles version numbers (X.Y.Z as opposed to X.Y) also caused some issues.
- Deployment of the whole scenario takes around 5 minutes, with the sudo installation being the biggest bottleneck.
- We did not make use of Ansible roles at this time, but none of the procedures involved here require the use of templates.
- Playing out the whole scenario takes a few minutes at most when familiar with the chosen procedures.
- Attackers are expected to find out about the other machines by performing network scans once they become a superuser on machine *bear*. Due to the flat structure of the network of this particular scenario (i.e. all machines share the same subnetwork), this operation is fairly trivial.

This scenario was designed and deployed as a proof of concept of URSID's capabilities and gave us a clear target for our initial procedures. However, due to the way URSID works, this work is still reusable if one wishes to deploy a different scenario. In fact, any scenario using the techniques involved in this APT-3-inspired example can now be deployed using URSID, as long as a) the scenario is defeatable at the procedure level, and b) the backtracking refinement scenario is able to find a working combination of procedures. Thus, any development work on procedures - even if initially intended for a single example - becomes part of the URSID's procedural database, providing even more options for future deployments. Furthermore, scenario designers do not need to worry about these technical

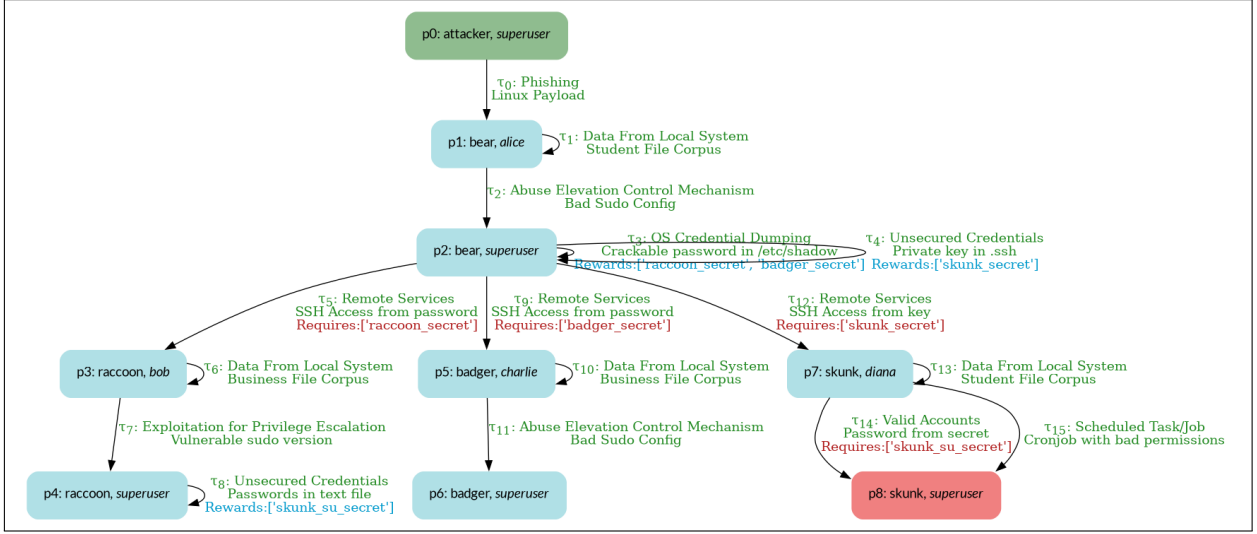


Figure 4.5 – APT3 inspired scenario on a procedural level deployed by URSID.

details of procedural implementation if they just want to deploy a scenario: they can design it at the technical level and let the procedural refinement process do the rest. We argue that these two points - the ability of URSID to reuse previous work and the ability to deploy scenarios without having to worry about technical implementation details - are the main advantages of URSID over the current state of the art in vulnerable architecture deployment.

4.3 Conclusion.

This chapter covered the conversion process between the configuration file issued from the procedural refinement, and actual virtual machine configuration files in the form of Ansible playbooks. We detailed how information about operating systems, software, accounts and files was parsed and converted, as well as how URSID handles networking, ending with a discussion on the deployment of an APT-3 inspired attack scenario.

This part of the URSID development was mostly technical, with the primary goal of getting a proof-of-concept deployment pipeline to play our instances locally. However, we tried to make this work modular in order to reduce the implementation costs of future constraint conversions. For example, our use of the CPE format to describe and convert OS and software versions, as well as the ability to add custom scripts for non-standard software installations, made it easier to implement new procedures in later deployments

Transition	Implementation	Expected execution
τ_0 : Phishing Linux Payload	Remotely detonable bash script	Execute provided script
τ_2 and τ_{11} : Abuse Elevation Control Mechanism Bad Sudo Config	/bin/less is executable as sudo by anyone	Launch shell from /bin/less
τ_3 : OS Credential dumping Crackable password in /etc/shadow	Passwords are weak	Dump /etc/shadow then use password cracking tool
τ_4 : Unsecured Credentials Private key in .ssh	Private key added to .ssh	Access .ssh
τ_5 and τ_9 : Remote Services SSH Access from password	SSH key matches the one obtained earlier	Reuse SSH key
τ_7 : Exploitation for Privilege Escalation Vulnerable sudo version	Sudo 1.8.27 + specific sudoers config	/bin/sh -u#-1
τ_8 : Unsecured Credentials Passwords in text file	Password in important.txt	Open the file.
τ_{12} : Valid Accounts Password from secret	Password matches the one obtained earlier	Reuse password
τ_{15} : Scheduled Task/Job Cronjob with bad permissions	Cronjob is editable and has sudo privileges	Edit cronjob to open a sudo shell

Table 4.1 – Procedures deployed with URSID as part of the APT 3 inspired scenario.

using URSID. Our use of architectural constraints at the procedural level was also relevant when unforeseen complexities arose in practice, such as installing software on older operating system versions: constraints gave us the ability to prohibit these interactions in the first place, rather than complicating the installation process further. Finally, the combination of Vagrant, Ansible, and VirtualBox, all used natively by URSID, made it easy to deploy and test instances locally. In addition, most of the implementation work is related to Ansible, which is responsible for deploying the virtual machines and therefore contains the information about the procedures to be implemented. This means that this Ansible output can be reused as part of other deployment stacks that do not rely on Vagrant and VirtualBox, which will be relevant in later experiments.

Apiculteur has been used as the basis for URSID deployments since its original development as part of a student project, while being constantly refined and improved to meet our new deployment goals and improve its modularity. The APT-3 inspired scenario was refined and deployed as part of a research seminar in January 2023. This piqued the interest of other researchers and graduate students within the team, who saw how URSID could help them in their own work. This culminated in the design, deployment, and execution of a live educational Capture-The-Flag experiment at a local university called CERBERE. The results of this experiment will then lead to URSID being used

as the basis for a challenge at a large French CTF event (BreizhCTF), which we named Casinolimit. This next chapter will describe these two experiments, discuss the technical challenges encountered, and show how URSID's unique capabilities were beneficial from both a research and educational standpoint.

PRACTICAL EXPERIMENTS USING URSID.

This chapter discusses two experiments conducted during this thesis that used URSID as the basis for deploying a vulnerable architecture. The first of these experiments was CERBERE¹. Conducted in April 2023 as part of a week-long cybersecurity curriculum at a local university, this challenge was designed to expose a group of students to blue and red team work in a professional cyber environment. The second experiment was Casinolimit², which took place as one of the challenges of the May 2024 edition of BreizhCTF, a government-sponsored capture-the-flag exercise. Both of these experiments used URSID for the generation process, but were also the result of collaboration between several researchers and engineers. We will describe these two experiments, each in its own subsection, with a particular focus on the role of URSID within these experiments. We will discuss the implementation challenges encountered, and show how URSID was beneficial to these experiments from both a research and a practical standpoint. The CERBERE experiment lead to the publication of a collaborative paper Besson et al., *op. cit.*, which goes over the whole experiment and its result but with a lesser focus on URSID itself.

5.1 CERBERE experiment.

5.1.1 Goals, challenges and design choices.

A team seminar presentation centered about a prototype version of URSID raised interest from a fellow PhD student, whose thesis Romain Brisse, « Exploration recommendations for the investigation of security incidents », Theses, CentraleSupélec, Feb. 2024, URL: <https://theses.hal.science/tel-04594042> dealt with the topic of automated

1. Cybersecurity Exercise for Red and Blue Team Entertainment.

2. Named after the fake casino in which the scenario takes place.

log analysis for better defense purpose. This discussion lead to talks about designing and deploying a scenario in order to gather attacker logs as part of an experiment. Other researchers quickly joined the project, with one interested in the harvest of network logs from attackers Helene Orsini et al., « AdvCat: Domain-Agnostic Robustness Assessment for Cybersecurity-Critical Applications with Categorical Inputs », *in: BigData 2022 - IEEE International Conference on Big Data*, Osaka, Japan: IEEE, Dec. 2022, pp. 1–13, URL: <https://inria.hal.science/hal-03893496>, and another one with the gathering of attacker actions within the context of a web browser Natan Talon et al., « SCWAD: Automated Pentesting of Web Applications », *in: Proceedings of the 21st International Conference on Security and Cryptography - Volume 1: SECRYPT*, INSTICC, SciTePress, 2024, pp. 424–433, ISBN: 978-989-758-709-2, DOI: 10.5220/0012721000003767. From this thesis point of view, this experiment seemed hugely beneficial for URSID, as we were interested in having a proper use case and demonstration of one of its most interesting trait: its ability to deploy slight variations of a scenario based on a single description.

This challenge was to be played at a local university as part of a week-long introductory cybersecurity program. Participants (mostly master’s level students, but also some faculty) were expected to have varying levels of hacking skill, but all were expected to be familiar with computer science and have a basic understanding of cybersecurity concepts. Note that the challenge was voluntary, which may have played a role in selecting participants who were interested, or at least curious, about capture-the-flag challenges. The number of participants would be known a week before the challenge, but was expected to be around a dozen. We expected to have a two-hour window to run the challenge. Given this short time frame, the fact that participants were expected to use their own machines to perform the attacks, our desire to collect attacker logs, and due to fairness concerns - since the host machine is able to access any of the deployed instances with root privileges through Vagrant - we decided that it would be unreasonable to ask participants to download the URSID and deploy the instances themselves. This made it desirable to generate and deploy the scenarios ourselves on our own machines, and provide some form of remote access to the participants.

To make the experiment more interesting for the other researchers, and to give the participants a more varied experience that would also be representative of a real-world scenario, we split the experiment into two parts. First, one team of participants - the red team - would attack instances of the scenario, generating browser, network, and system data. Then, another team of participants - the blue team, ideally made up of people who

did not participate in the first part of the experiment - is given the attack logs generated by the first part. These participants are expected to analyze the attacker's path and actions taken during the first part of the experiment. Each part would last one hour. We will mostly discuss the red team portion of this experiment in this chapter, as it is the portion in which URSID had the most involvement.

We identified the challenges of the scenario and red team part of the experiment as follows:

- The scenario should be defined on a technical level in order to make proper use of URSID's high level scenario description.
- Each technique within the scenario should ideally offer several procedures, in order to showcase URSID's refinement process and abilities of deploying different instances of a single technical scenario. This also would have educational benefits, as students would be less able to copy on each other.
- Parts of the scenario should have a web component accessible through a web browser. This exploit should require a sequence of actions on the web browser itself, as well as a discovery process in order to provide interesting browsing attacker data. Note that some web exploits which lead to direct access on the machine - such as a reverse shell - are no longer trackable through a web browser once they are accomplished.
- Both networks and system logs should be harvested on every machine for each instance and be gathered at the end of the experiment for later analysis.
- Instances should be made available remotely, while still being secured or hard to find for non-participants, in order to avoid background noise due to automated bot scanning attacks.
- Our deployment solution should be able to handle at least a dozen instances (one for each participant), and have some backups in case something fails.
- Difficulty of the challenge should be adapted to the available time slot as well as the expected skill of the participants.

The chosen scenario is available in figure 5.1. This scenario offers three machines: *zagreus*, *hades* and *melinoe*³. First, despite having access to *hades*, the attack must first go to machine *zagreus*. This is because all the transitions they make take in order to access *hades* have secrets required in order to be performed, which have to be first gathered on

3. Named after Greek mythology figures, keeping in theme with CERBERE.

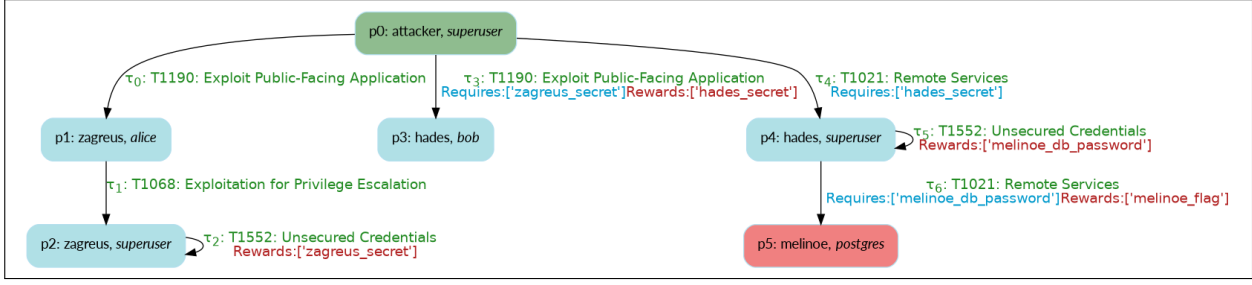


Figure 5.1 – CERBERE scenario on a technical level.

zagreus. Attacker may acquire a position on *zagreus* by exploiting T1190: *Exploit Public-Facing Application*. Note that this technique itself does not specify that this part of the scenario will involve a web application: this fact will instead be apparent when we discuss our choices of available procedures for this scenario. Once this initial footing is acquired, the attacker may perform an exploit (T1068: *Exploitation for Privilege Escalation*) in order to acquire a privileged attack position with administrative rights on this machine. Once this is done, they may harvest credentials from this position by performing T1552: *Unsecured Credentials*, which provides them with the secret `zagreus_secret`. This secret can then be used in order to perform T1190 again on machine *hades*, which rewards them with the secret `hades_secret`. Using this secret, they acquire a privileged attack position on this machine (using T1021: *Remote Services*), before extracting credentials from this position (T1552: *Unsecured Credentials*). Finally, using this `melinoe_db_password` secret, they can reach the winning position (*melinoe, postgresql*), thereby beating this scenario.

This scenario - with 3 machines, 5 attack positions⁴ and 7 transitions spanning 4 techniques - was judged to be beatable within the allocated time frame for the more skilled participants. We also inferred that the number of steps involved would translate to less successful attackers still being able to complete parts of the scenario.

On a procedural level, we managed to get variability on each of the techniques available in the scenario. This did not translate into each transition having variability between instances, due to the way secrets interact with the refinement process. We will now detail our list of procedures available for each technique.

T1190: *Exploit Public-Facing Application* For transition τ_0 leading to (*zagreus, alice*), the chosen procedure should require no prior secret knowledge and not reward

4. Not counting the attacker's machine (*attacker, superuser*).

any either. Furthermore, we wished for that transition to involve a web application for research purposes, as it is the first transition expected to be taken by the attacker. Indeed, in the event where the web related procedures were only at the end of the scenario, and we misjudged the skill of the attackers and the difficulty of the scenario, we could have ended up with little to no web-based attacker data. In order to still present some procedure variability despite those restrictions, as well as being able to offer different levels of difficulty for the participants, several variations of a web application, leading to a reverse shell with varying levels of difficulty, were designed. For transition τ_3 (leading to $(hades, bob)$), a procedure (π) that both required and rewarded a secret was needed. We ended up designing another web application, but due to time and development constraints did not build any variations for this one. Thus, out of the three associated procedures to this technique π_1 , π_2 and π_3 , due to our secret constraints τ_3 will always be associated with π_3 , while τ_0 will have variation between π_1 and π_2 .

- π_1 : *Website with command injection (easy)*. This web application - built using the Node.js runtime environment - is an image database in which users may create accounts and upload their own images. However, one of the search bars of the website is flawed and can be exploited in order to execute bash code as the user hosting the website. Once this vulnerability is found, an attacker may use it in order to acquire a session as that user - for instance by establishing a reverse shell Internal All The Things, *Reverse Shell Cheat Sheet*, URL: <https://swisskyrepo.github.io/InternalAllTheThings/cheatsheets/shell-reverse-cheatsheet/>. This procedure requires Ubuntu 20.04⁵, the `npm` software to be installed and running on port 3000, the `curl` software to be installed, the `exit` user account to have an `npm` service, and website files to be copied on the machine. Installation of the `npm` software required us to write a custom script as it is not directly available through the Linux `apt` utility, requiring an outside repository to be added first.
- π_2 : *Website with command injection (medium)*. This website is almost the same as the one used for π_1 , but the search bar containing the injection vulnerability now has a character limit of 50. This makes it much more difficult to inject the reverse shell code directly in this search bar. Instead, attackers are expected to make use of the image upload functions of the website, as it lets them upload any file as long as it has a `.jpg` extension. Attackers may upload their script this way, then change script

5. Technically would work on other operating systems, but implementation issues lead us to limit ourselves to this particular OS.

permissions and execute it through the code injection vulnerability, which is doable within the character limit. Architecture constraints are the same as π_1 , expect the website files to be copied are slightly different.

- π_3 : *Django directory traversal rewarding SSH key.* This website - ran using the Django framework - has an administrative page accepting a login and password combination. If successful, the `/notes` directory of the website may be accessed, which contains a SSH key. This directory can either be found through traditional directory traversal - i.e. brute-forcing commonly used directory names until a match is found - or by carefully analyzing error messages given by the website itself when invalid directories are accessed. This procedure requires Ubuntu 16.04, 20.04 or Debian 11, a python software to run on port 8000, the `exit` user to run a Django service, files for the website to be copied, executable files in order to add a user to the website (interacts with secrets) and files related to the SSH key (interacts with secrets). Note that when executing the procedure attackers are not truly in possession of the attack position in charge of the website, they are only interacting with the website itself. This is a weakness in URSID's representation choices.

T1068: *Exploitation for privilege escalation* This technique is used for the transition τ_1 , which has no secret requirement or rewards. This gives us more liberty in term of exploits to provide the attacker with, as those can be pretty much any privilege escalation CVE we can reasonably install. We ended up offering two different exploits for this technique.

- π_4 : *Vulnerable sudo version.* Linux systems make use of the `sudo` package to allow users to run programs with the same permissions as other users, in particular privileged users. These permissions are configured in the `sudoers` file of the machine, and may in particular allow a command to be executed by any user except a specific one. CVE-2019-14287 NVD, *CVE-2019-14287 Detail*, 2019, URL: <https://nvd.nist.gov/vuln/detail/CVE-2019-14287> (visited on 08/30/2023) allows attackers to bypass this restriction on some versions of the `sudo` package by invoking `sudo` with a crafted user ID, which may lead to them acquiring a shell as a privileged user. This procedure requires Linux, the `sudo` package to be installed at version 1.8.27⁶, the `libpam0g-dev`, `make` and `gcc` packages in order to install `sudo`, specific

6. While the NVD report indicates that this exploit should work on any version of `sudo` up to 1.8.28, we found that in practice this specific exploit was less consistent on older versions.

edits to the `/etc/sudoers` file and the `exit` user to have superuser privileges. The `sudo` package requires its own script to be installed. Sudo versions are converted into floats (1.8.27 $\rightarrow$ 1.827) for constraint reasons, before being converted back into actual version numbers.

- π_5 : *Vulnerable pkexec process*. The `pkexec` utility is a native Linux command which allows a user to execute a program as another user. Older versions of this software however are vulnerable to CVE-2021-4034 *idem*, *CVE-2021-4034 Detail*: Attackers may abuse an oversight in the way this utility handles parameters counts in order to execute arbitrary code. This may lead to the attacker obtaining administrative permissions on the machine. Scripts performing the exploit are easy to find online. While installing the `pkexec` utility at a specific (vulnerable) version is nontrivial, we found that the Vagrant box for Ubuntu 16.04 was natively vulnerable to this exploit, and chose to use that as a constraint instead. This procedure thus only requires an Ubuntu 16.04, and the `exit` user of the transition to have superuser privileges.

T1021: Remote Services Both the transitions linked to that technique in the scenario (τ_4 and τ_6) deal with secrets, but they each have different requirements and rewards. We thus provided a procedure for both of those cases: one website - for web attack research purposes - and one SQL database, in order to increase the variety of the type of services available in the scenario.

- π_6 : *SSH access from key*. This corresponds to an attacker using a previously found private SSH key in order to make a valid connection to a remote account. Internally, we generate a key pair as a secret, then take either the public or private key part when adding files to a machine. This procedure requires a Linux, an SSH service⁷, an SSH software running on the port 22, and files to be edited in order to add the public key corresponding to the private key to the `exit` machine. The SSH service also gets restarted at the end of the deployment process in order to account for any possible new configuration.
- π_7 : *SQL Server rewarding a flag*. This database is accessible remotely through a password via a PostgreSQL service. Once logged in, attackers may explore the available tables within the database until they find one containing the flag indicating that they won. This procedure requires a password secret and rewards a flag secret. It also requires Linux, the `entry` user to be a superuser, the `exist` user to run a

7. Available by default on most Linux distributions

PostgreSQL service, files related to that database to be copied and an executable file interacting with secrets, which will be used to add the flag to the database.

T1552: *Unsecured Credentials* All of those procedures deal with secrets. Furthermore, we only implemented procedures which dealt with plaintext passwords (as opposed to SSH keys for instance). This is due to the fact that both π_3 and π_7 are related to passwords as well. While we could have implemented procedures for T1552 with other types of secrets, they'd have never been picked due to our secret preconditions system. We kept the difficulty fairly low in these procedures due to the attacker's expected time constraints.

- π_8 : *Password in txt file*. This file (stored in the home directory of the exit user) is plainly named important.txt and contains passwords that a user would have written in order to remember. This procedure requires Linux and to create a file which interacts with secrets. This procedure rewards plaintext passwords.
- π_9 : *Password in bash history*. Linux systems using bash as a shell save by default previously entered commands of the user in a file known as the bash history. If a user ends up entering a password in their shell - possibly by accident -, that password would then be stored in plain text in this file and easily accessible. The bash history is a hidden file by default, making this procedure slightly less trivial than π_8 . This procedure requires Linux⁸, and to create a bash history file which interacts with secrets. This procedure rewards plaintext passwords.

The fact that there are several types of services possible - such as shell, website or SQL - caused some issues during development, especially for the T1190: *Exploit Public-Facing Application* and T1021: *Remote Services* techniques. We circumvented this problem by only implementing a select amount of procedures, which combined with the secret precondition system gave us control over the type of service used for τ_0 and τ_6 . We did consider adding our own subtechniques to the ATT&CK matrix, which would make it possible to keep the refinement process but make it possible to restrain the range of procedures a given transition can take. For instance, this could result in a new subtechnique T1190.001: *Exploit Public-Facing Application - Web Application*, which would guarantee the use of a web related procedure. We chose not to do this at the time in order to keep ourselves aligned with the official ATT&CK matrix, but this may be desirable in the future. Furthermore, some of the procedures do not grant the attack position (as in a shell) per-se, but rather

⁸. Technically not all Linux distributions use bash natively, but all the ones we consider in the deployment do.

access to a service (such as a website) hosted by the user of this attack position. This is a weakness of our representation but can be worked around during the scenario design. In CERBERE's case, this is relevant for transitions τ_3 (leading to $(hades, bob)$) which is access to a django website and τ_6 (leading to $(melinoe, postgres)$), access to a postgresSQL database. As neither of these positions lead to new transitions being available to the attacker - with one providing them with a secret and the other acting as the goal of the scenario -, this representation issue does not cause any discrepancies with an attacker's actual options.

The procedural refinement of this scenario is fairly straightforward due to the low amount of procedures implemented. The main changes between two instances are the difficulty of the website hosted by p_1 , the privilege escalation between p_1 and p_2 , and the way credentials are harvested in p_2 and p_4 . All the secrets and their values (in this case passwords and SSH keys) are also unique for each generation. We ranked the difficulty of each of the possible procedures in order to know the relative difficulty of a given generation. This was done in order to provide the more skilled students with harder instances, in the event in which we were familiar with their skill level beforehand. Fifteen different instances were generated in total, providing enough scenarios for each of the dozen expected participants as well as some backups.

These scenarios were originally developed, deployed, and tested locally, with the machine using the URSID also hosting all the virtual machines. However, this setup would not be sufficient for the experiment, as it was necessary to provide remote access to the machines. Our solution was to use a combination of network rules, a custom DNS server, and a gateway host used by the participants to connect to the Internet. All 15 URSID instances were created on a single machine with 36 cores and 80 GB of RAM. These instances were then made available to the attendees through a personalized URL. Thanks to our DNS setup, this URL would point back to their designated instance on our local network on a specific port, to reduce the chances of automated attackers finding our machines through mass Internet scans. Two of the three machines - *zagreus* and *hades* - were made available this way, while *melinoe* required the attacker to be inside the network in order to be visible. This part of the deployment was not handled automatically by URSID and had to be implemented manually. This implementation work was done in collaboration with an inria engineer, who also provided the hosting machine. We met regularly to ensure that our local deployment of URSID was compatible with their custom network setup, and to discuss the architecture of URSID as a whole to improve its reusability and efficiency. For

instance, the work on CERBERE is what lead to the use of Ansible roles as part of the generation, which we described in part 4.

Figure 5.2 showcases the overall architecture of the CERBERE experiment. URSID handles the generation of every scenario and their related instances, represented by **Scenario i** on this image. The red teams attempt to exploit all 3 machines of their specific instance through our provided gateways. Their system actions were recorded thanks to our installation and configuration of auditd on every instance, which was done using specific architectural constraints during the generation process. Web browser logs are acquired through a customized browser extension installed with a docker application, while network logs are acquired through the use of an extra machine using Suricata Suricata, URL: <https://suricata.io/> (visited on 07/05/2024). This machine, as well as the other machines which acted as network gateways for each of the instances, were not part of the attack scenario, and as thus were not represented. Finally, each generated instance had 2 more machines which implement the GHOSTS Carnegie Mellon University, URL: <https://github.com/cmu-sei/GHOSTS> (visited on 07/05/2024) framework in order to generate fake traffic on the network. Meanwhile, the blue team participants had access to the attack logs generated by the red team in the previous part of the experiment. They were helped in this investigation by a recommender system for analysts which was part of another researcher's work Brisse, *op. cit.*

5.1.2 Reflections on URSID's use as a CTF generation platform

CERBERE was URSID's first proper use as a CTF generation platform, and as such gave us valuable insights in the current strengths and weaknesses of the tool. In particular, we will now evaluate how much automation and ease of use URSID provided at the time in the context of scenario design and deployment, and which parts of the process warranted improvement.

First, adding new procedures was greatly facilitated by the tool's high-level/low-level paradigm. Whenever new procedures were designed, for example by creating different versions of the web pages, once the description of that procedure in terms of architectural constraints was added to the URSID library, it was automatically added and eligible for the refinement process, resulting in generated instances being able to be vulnerable to that procedure. This also natively increases the variability of output scenarios thanks to our refinement process, with no additional work required on this part once procedures are added. Adding software by default on each machine - to add logging capabilities,

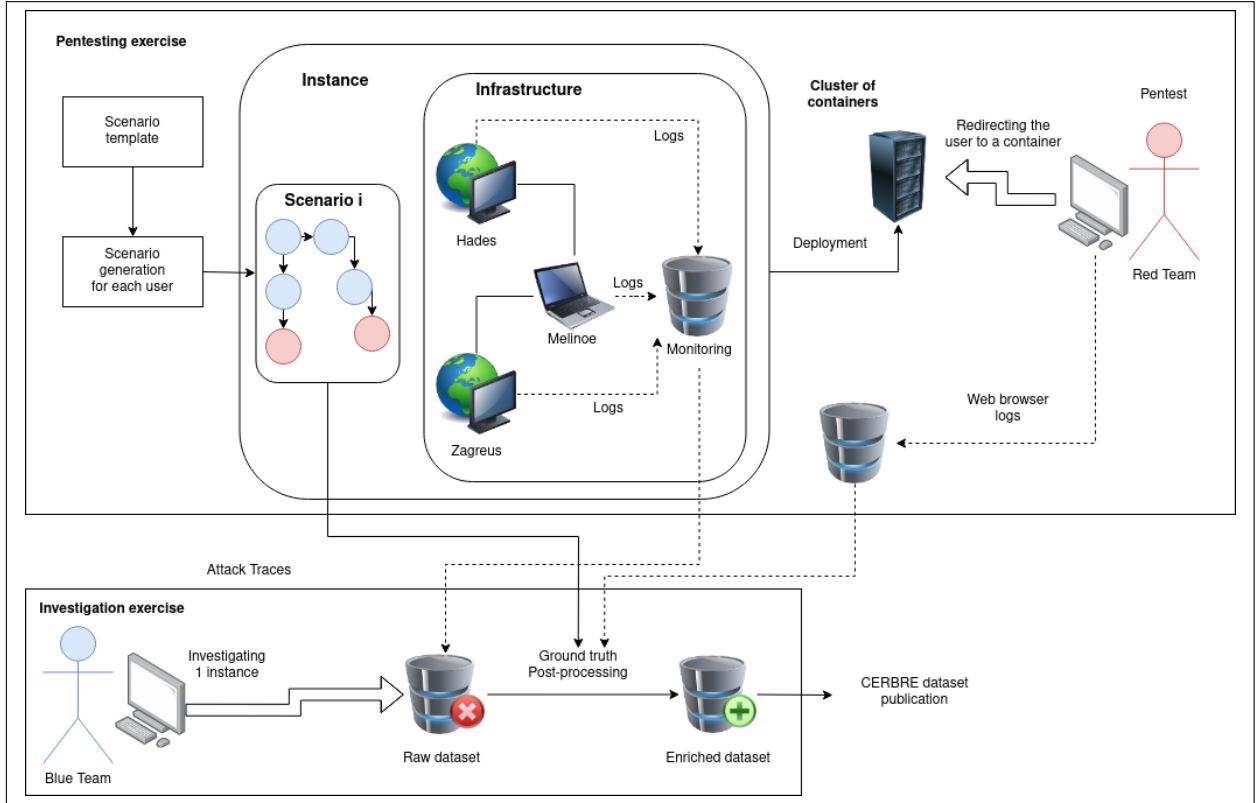


Figure 5.2 – Overview of the CERBERE experiment.

for example - was also a simple task, requiring only that specific procedures containing the relevant information be added by default at the beginning of the refinement process. Adding variability to output instances was fully automated and required no additional work, thanks to the overall design of URSID. In addition, some collaborative aspects of scenario design were already facilitated. For example, updates to files associated with procedures - such as web pages - could be done independently by other researchers and still be used by their corresponding procedure. This made it easier to divide the work of scenario design into multiple subtasks. Finally, while the URSID output at the time was designed for deployment over a local network using a combination of a Vagrant file and Ansible playbooks, these playbooks were ultimately successfully used as the basis for our remote deployment, as they contained all the information needed to provision the machines.

However, the fact that URSID does not natively handle remote deployment has been a problem throughout development. First, it makes the experiment more difficult to

reproduce, since the work and code resulting from this deployment was done independently of URSID and is not available with the tool itself. The overall development and testing workflow also needed improvement: instances were first generated and tested locally based on the URSID output. This output was then fed into a different deployment pipeline for our remote instances, which required another round of testing. Some procedures also required additional work to implement beyond writing their architectural constraints. For example, the installation of certain software (such as `sudo`) was non-trivial and required the addition of some code. Efforts to automate this process, in order to minimize the amount of code editing required to add new procedures, were identified as an important future improvement to URSID, in order to make it as easy as possible for other users to use.

Thus, while URSID was arguably still somewhat immature at the time, it was fulfilling its function of facilitating variability in the design and deployment of vulnerable architectures. Despite the problems identified, it was possible to go from designing a scenario at a technical level (through the provided GUI) using already implemented techniques, to refining and deploying it after a brief explanation. Such demonstrations were performed several times during this thesis to show the tool to other researchers.

5.2 Experiment results.

5.2.1 Course of the experiment.

The final turnout for the experiment was of 13 participants for the red team part and 9 for the blue team part. These consisted mostly of students coming from a computer science background, but also included a few of their professors. Participants were given an hour for each of the sections of the experiment⁹. Participants were instructed to start with machine *zagreus* using their given URL, and pivot to machine *hades* once they find an explicit reference to that machine within the scenario. This corresponds to participants finding out the credential stored on the (*zagreus*, *superuser*) position, which mentions machine *hades* by name alongside providing the required credential. Due to our network setup, initiating a reverse shell on machine *zagreus* could not be done directly from the participant's machines, as that IP was not routable. This is not an issue when playing out the CERBERE scenario on a local deployment. We thus made the choice of specifically instructing participants to

9. We did go a bit over the allocated time, but this was partly due to technical issues with docker installations on the participants' machines.

use ngrok Ngrok, URL: <https://ngrok.com/> (visited on 07/05/2024) as a tool in order to facilitate this step. These instructions detailed how to install and use the tool in order to use a ngrok server as a bridge between machine *zagreus* and their own, in order to make a reverse shell possible.

Finally, we made the choice of providing struggling participants with occasional hints in how to proceed to the next step. We did not however provide direct solutions to their issues, but rather nudged them in the right direction of what they needed to do in case they were stuck on a step for too long. This was particularly relevant for the first step of the scenario, as the amount of red herrings available on the website hosted by *zagreus* made it likely for less experienced participants to get lost in it and miss the page containing the command injection¹⁰. This was done in order to improve the educational value of the experiment, as someone getting stuck on the website for the whole duration with no hints would have made for a poor learning experience. However, as will be discussed in the next section, this did not result in most attackers beating the scenario, as we were careful not to give the answers away directly.

Some of the participants had to be provided with new instances, as their attempts of initiating a reverse shell on machine *zagreus* lead to the website service shutting down. However, we had enough back-up instances to provide them throughout the whole experiment. Most of the participants did not complete the experiment in the expected time. We identified several reasons for this:

- The reverse shell on website *zagreus* was a significant challenge for the average skill level of the participants, as the website contained a lot of functionalities which were ultimately useless for the exploit. Furthermore, unlike some of the other procedures - which consisted mostly of a single step such as "open the file containing credentials" or "exploit a specific CVE" -, exploiting this procedure required a) identifying the location of the command injection b) setting up a network bridge c) successfully input code leading to a reverse shell via their network bridge.
- Tracking of attacker's browser activity required them to run a specific docker container, but a lot of the machines participants brought did not have docker installed, and some had trouble installing it. This issue is not linked to URSID's involvement in the experiment, but did lead to significant time losses for some participants.
- Access to the network was done through a reverse shell initiated in machine *zagreus*.

10. Anecdotally, this command injection is very easy to find using automated pentester tools such as Burp Suite. However, most participants were not familiar with those.

Depending on the reverse shell script used, this session may be unstable or difficult to use. For instance, the most commonly used bash script leading to a reverse shell would not all users to use shortcuts such as CTRL-C/CTRL-V (for copy and paste) or keyboard arrows (to reuse previous commands). Combined with the inherent challenge of the scenario itself, a participant mistake in any of the steps may lead to them losing their reverse shell, which in turn would force them to reiterate some of the attacks they already successfully executed (such as the reverse shell or the privilege escalation on *zagreus*). This caused additional time losses for some of the participants.

Anecdotally, participants appreciated the scenario variability. Upon landing on position (*zagreus*, *superuser*), one of the participants promptly tried to find a .txt file containing the required credential, as it was what their neighbor did. However, since their own instance had a different procedure (and upon being informed that such a discrepancy was normal), they managed to find the credential stored in the bash history on their own.

5.2.2 Results analysis.

Thanks to our logging setup, which included the harvest of auditd system logs on every single machine, we were able to identify attacker progress within each instance of the scenario. Indeed, these logs would show actions such as interactions with a specific file (which is useful in order to check attacker gathering credentials) or opening a session as a specific user (useful for checking access to new attack positions). Note that our logging setup also included network and attacker browser data, which were particularly relevant for the other researchers involved in this experiment. This thesis will however put its focus on the analysis of system logs. This choice was made for two primary reasons. First, the installation and configuration of auditd was handled by URSID itself through our constraint system. This is unlike the network logging - which required an additional *suricata* instance to be generated, which at the time was outside the scope of URSID - and the attacker browser logging, which was done through a customized docker container not generated by URSID. We also judged system logs to be the most relevant in our task of tracking procedure executions, as they are able to track every command entered within a shell generated for a given user.

While these logs could be theoretically be analyzed by hand, the sheer amount of

information contained in them¹¹, combined with the amount of machines and instances to analyze (3 files for each of the 15 instances) would make this task unpractical. We thus looked for solutions in order to automate our log analysis, and made the choice of using SIGMA SigmaHQ, URL: <https://github.com/SigmaHQ/sigma> (visited on 07/05/2024). Describing itself as a "generic and open signature format that allows you to describe relevant log events in a straightforward manner", SIGMA allows its user to write and share detection rules adapted for log files analysis. These rules correspond to exploits or procedures that analysts may write and share, which parse log files and look for patterns such as specific commands being entered or programs being launched. The choice of this software was made for several reasons:

- SIGMA is widely used in the industry and is natively compatible with log processing engines such as Splunk *pySigma Splunk Backend*, URL: <https://github.com/SigmaHQ/pySigma-backend-splunk> (visited on 07/10/2024) or ElasticSearch *pySigma Elasticsearch Backend*, URL: <https://github.com/SigmaHQ/pySigma-backend-elasticsearch> (visited on 07/10/2024).
- SIGMA rules can be shared between users, leading to huge rule compendiums being publicly available *SIGMA detection rules*, URL: <https://github.com/mdecrevoisier/SIGMA-detection-rules> (visited on 07/10/2024); *Sigma - Generic Signature Format for SIEM Systems*, URL: <https://github.com/SigmaHQ/sigma> (visited on 07/10/2024), sometime grouping them by ATT&CK techniques.
- SIGMA supports auditd Linux logs.
- SIGMA and its rules may be interacted with using Python, which is the language used by URSID and with which this thesis' author is the most familiar with.

We thus made use of Zircolite *Zircolite - Standalone SIGMA-based detection tool*, URL: <https://github.com/wagga40/Zircolite> (visited on 07/10/2024), a Python SIGMA-based detection tool which includes auditd support, in order to implement Sigma rules for URSID. Zircolite is able to convert auditd logs into databases, which can then be interacted with using SQL written rules. It then outputs a list of all rules that were matched for a given dataset. Figure 5.3 showcases an example of such a rule, linked to the exploitation of CVE-2019-14287 which corresponds to procedure π_4 in our CERBERE experiment. This rule indicates the name of the procedure, which ATT&CK technique it

11. Around 500 Ko of text files for each machine of each instance, with each system call generating 5 lines of text on its own.

```
1 {
2   "title": "Exploitation for privilege escalation: Vulnerable sudo
      ↪ version (CVE-2019-14287)",
3   "id": "1",
4   "description": "CVE 2019-14287",
5   "tags": [
6     "attack.privilege_escalation",
7     "attack.t1068"
8   ],
9   "falsepositives": [
10  ],
11  "rule": [
12    "SELECT * FROM logs WHERE ((type = 'EXECVE' )) AND (a1 LIKE
      ↪ '-u#-1%' OR a1 LIKE '-u#4294967295')"
```

Figure 5.3 – Example of zirconite rule, written for the procedure linked to (CVE-2019-14287).

corresponds to, and functions by looking for specific commands executed in a shell which are linked to the exploitation of this particular CVE.

We thus wrote a zirconite rule for every procedure implemented in CERBERE, except for π_3 : *Django directory traversal rewarding SSH key*. This procedure takes place entirely on a website, and actions performed by attackers there were not directly detectable through auditd system log analysis. We then matched these rules with the dataset collected during the CERBERE experiment grouped by instances. This provides us with a list of detected procedures for each of the instances played during the scenario. We finally compared this information with the procedural level scenario description of each of the deployed instances, letting us know which and when transitions were taken for this specific instance. Figure 5.4 showcases one of those results for instance 4. The red arrow and their timestamp correspond to the first time we detected this attacker performing this transition. This gives us the insight that this specific attacker took around an hour to finish the scenario after performing the first transition.

We were not able to detect every attacker procedures this way. For starter, transition τ_3 was always linked to procedure π_3 , which we were not able to detect through system log analysis. Furthermore, attacker sometime performed transitions in ways we did not

expect and were not caught by our detection rules. However, due to the linear nature of the scenario as well as its representation in our formalism, we are able to deduce the overall attacker path despite failing to detect some of the transitions. For instance, consider scenario 7, whose automated attack path analysis is represented in figure 5.5. In this particular instance of the scenario, we failed to detect the attackers performing τ_0 in order to gain access to *zagreus,alice*, due to them using a python command in order to download and execute a script. However, we are still able to detect the attacker performing the privilege escalation τ_1 and acquiring credentials in τ_2 . Performing these transitions indicates that the attacker had access to attack positions (*zagreus,alice*) and (*zagreus,superuser*), and since the only expected access to position (*zagreus,alice*) is τ_0 , we infer that attacker most likely performed that transition anyway. Furthermore, such discrepancies in attacker’s expected paths may point analysts to look into this specific machine’s logs more in details, which is how we found out about their use of a python command in the first place. This is also how we handled the previously mentioned issue of being unable to detect τ_3 : since this transition rewarded the secret `hades_secret`, which is required by transition τ_4 , we inferred that any attacker performing τ_4 must have necessarily performed transition τ_3 in the first place.

This automated analysis had to be done after the experiment was over, as our setup at the time was only able to gather logs after the red team part was complete. Ideally, we would have been able to gather logs alongside the experiment, and combining it with automated log analysis and our scenario formalism been able to track attacker’s path through the scenario in real time. This was considered but ultimately not implemented for CERBERE due to development time constraints, but was kept in mind for future experiments.

Table 5.1 showcases how many attackers were detected for each transition, both from our automated log analysis of the red-team part of the experiment, and from the work of the blue-teamers during the second part. As URSID was mostly relevant for the red-team part, we will focus our discussion on the results identified from our automated analysis.

First, we identify that the first transition τ_0 , which lead attackers to the attack position *zagreus,alice* was a major block for almost half of the participants. As previously discussed, this is due to the relative complexity of this step, which required chaining a few actions together as well as navigating a complete website full of red herrings. Attackers which were able to perform this step all managed to take transitions τ_1 and τ_2 , showing that these were by comparison easier to perform. The next big block was getting to the position

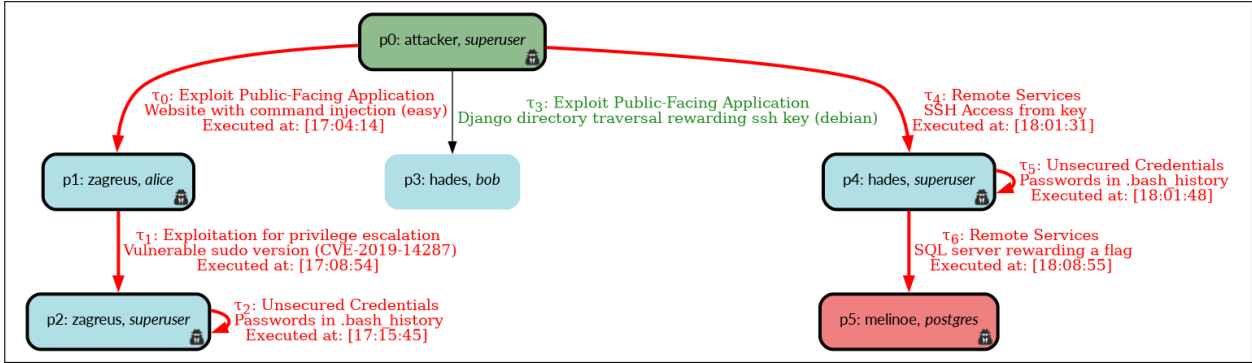


Figure 5.4 – Attack path detected through automated log analysis on instance 4, in which the attacker finished the whole scenario.

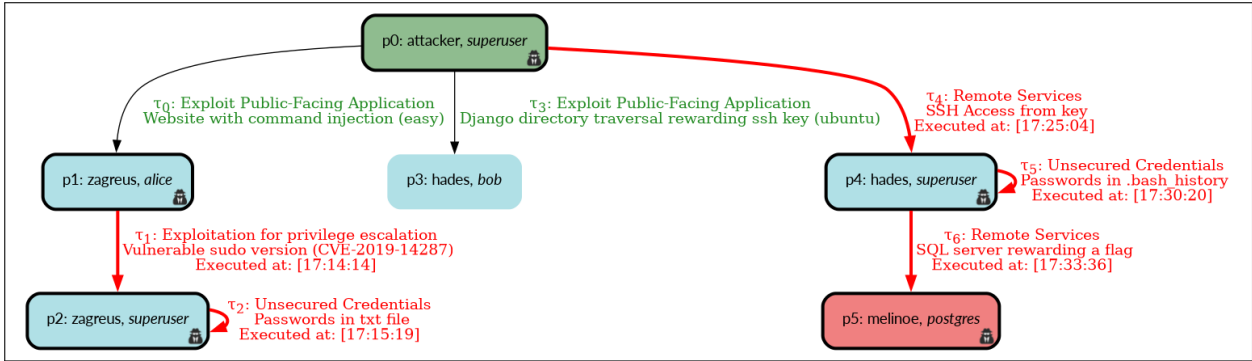


Figure 5.5 – Attack path detected through automated log analysis on instance 7, in which we failed to detect the first transition being taken.

(hades, superuser). While accessing position (hades, bob) was fairly trivial (only requiring attackers to enter the passwords they acquired on (zagreus, superuser) into a website login form), finding the position of the secret on that website, recovering it, identifying what it is useful for and using it successfully was difficult for some of the participants. Figure 5.7 showcases the credential attackers had to find through directory traversal on the /notes page of the website, which was hinted at through error messages provided by the site itself visible in figure 5.6. Once found, attackers still had to correctly format this information, paste it into their reverse shell and use it to initiate a ssh connection to the right user. Note that due to our limitation in detecting transition τ_{au_3} , it is possible that some attackers did acquire this credential but failed to use it, in which case they were not detected as having performed that transition at all.

		Red team exploitation	Blue team discovery
Total nb players		13	7
Scenario step	MITRE ATT&CK Technique		
T_0	T1190	7	5
T_1	T1068	7	5
T_2	T1155	7	3
T_3	T1190	4	2
T_4	T1021	3	2
T_5	T1552	3	2
T_6	T1021	3	0

Table 5.1 – Successful attack automated detections (Red team) and discoveries by participants (Blue team).

The results of this automated log analysis was part of the dataset we made public alongside our CERBERE paper, and is available online Pierre-Victor Besson et al., URL: <https://gitlab.inria.fr/cidre-public/cerbere-dataset/> (visited on 07/05/2024). This dataset contains:

- The aforementioned attack paths.
- The raw logs of red teamers (network and system), labeled with the corresponding MITRE ATT&CK techniques.
- The scenario itself, which can be deployed locally using URSID.

5.2.3 Conclusion of CERBERE experiment

In the scope of this thesis, CERBERE was intended as a proof of concept for URSID’s abilities to deploy vulnerable architectures, both for educational (familiarizing students with the work of red and blue teams) and research (gathering attacker logs for further analysis) reasons. We argue that it accomplished both of these goals, in particular leading to the publications of a paper Besson et al., « CERBERE: Cybersecurity Exercise for Red and Blue team Entertainment, REproducibility » and a public labeled dataset Besson et al., to go with it. We also published a working version of URSID alongside usage instructions for deploying CERBERE, increasing the value of this dataset as it is now reproducible.

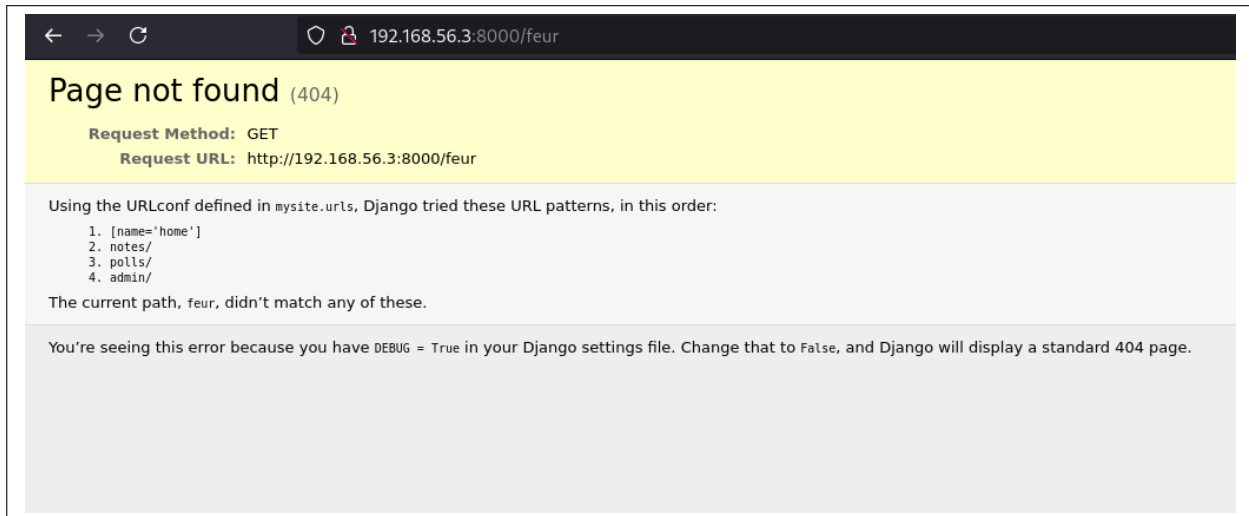


Figure 5.6 – Hint for credential location through an error message on the django instance. The `/notes` directory could also be found through a brute-force listing of directories.

This dataset was used internally by at least three other PhD students, for purposes ranging from training AI models to improving log analysis software. URSID made it easy to increase the variability of the instances used by offering different procedural variations of the same technical level scenario. These variations made the experiment more interesting for participants - making it harder to just copy your neighbor in both the red and blue team parts of the experiment - while also increasing the variety of attacks available in the published dataset. Now that each CERBERE-related scheme has been implemented, it is also easy to re-deploy an instance locally, or to tweak the experiment by adding new schemes or nodes. For example, a simplified version of CERBERE containing only the *zagreus* machine was successfully deployed as part of a university lab with minimal effort: all that was required was to create a new technical level scenario containing only that machine. The two main time-consuming aspects of development were the addition of procedures and our implementation of the specific network setup required for this experiment, but once this work is done, it is easy to tweak the experiment, which is the goal of URSID compared to the state of the art in vulnerable architecture deployment software.

As a first practical application of URSID, CERBERE was also an opportunity to develop, refine and identify weaknesses of the tool and the experiment itself. We identified possible improvements as follows:

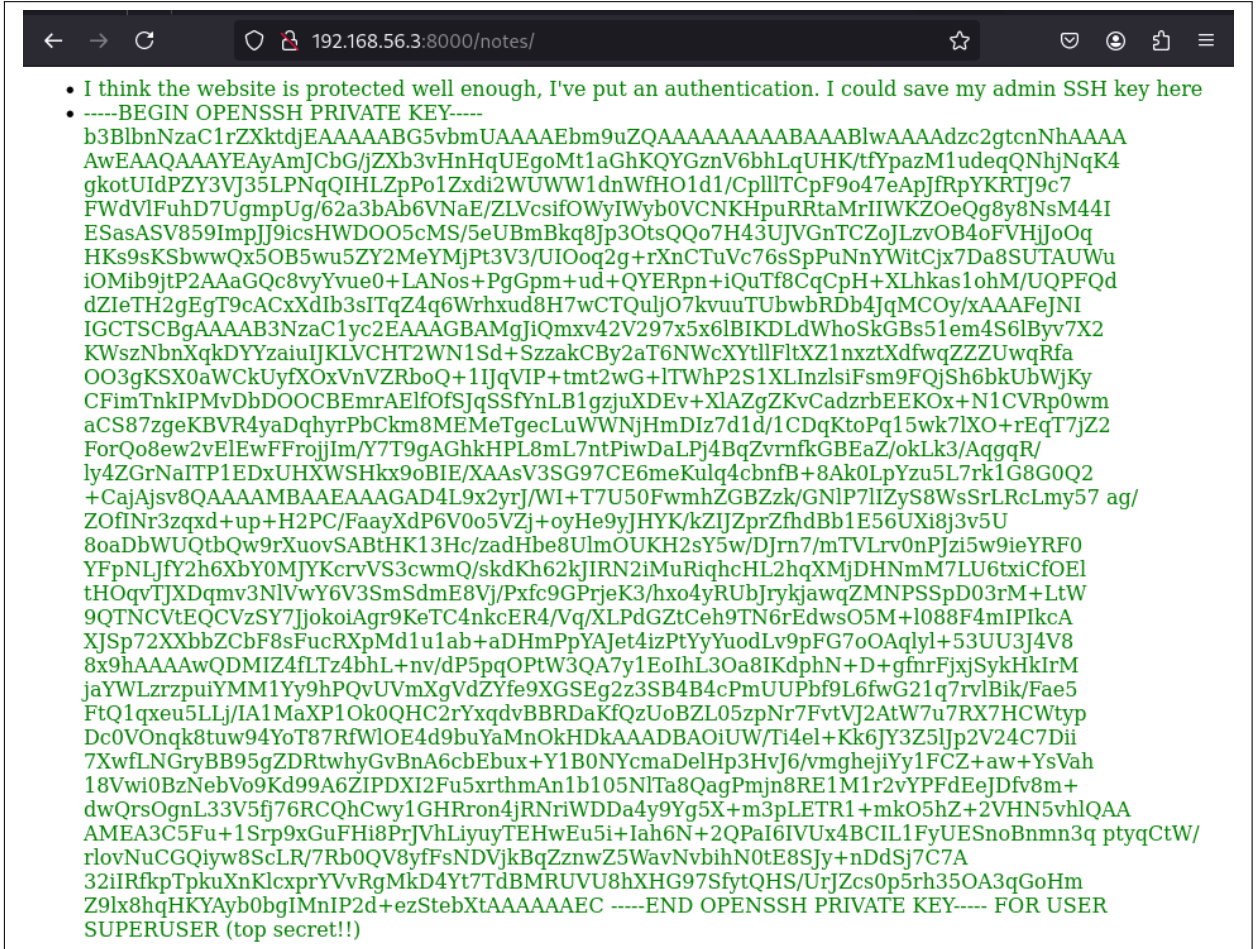


Figure 5.7 – Credential to be found as a reward for taking transition τ_3 , hidden in the /notes directory of the django instance.

- The networking aspect was still not handled by URSID outside local deployments, and had to be setup manually.
- The first step of the scenario was arguably too hard compared to the rest, which lead to a significant entry barrier for an educational exercise.
- Log analysis automation was not done live, meaning attacker paths could only be tracked after the experiment was concluded¹².
- CERBERE was a collaborative work, which combined development of URSID and its procedures, the websites linked to some of these procedures, and the networking and deployment parts. While this collaboration was overall successful, it also required a

12. Or by walking around the classroom and asking participants how they were doing.

lot of "band-aid" solutions - such as having to manually pull git repositories before launching the tool, or having to write buffer configuration files in order to convert URSID's output to another format for networking purposes. These solutions would not scale well for other projects.

We met regularly - in particular with the engineer who was charge of the networking and deployment part - in order to improve on these issues. However, the success of CERBERE quickly lead to another opportunity of using URSID for vulnerable architecture deployment, in the form of a challenge for BreizhCTF, a local Capture-The-Flag competition. This challenge, which we dubbed Casinolimit and used URSID as a basis for scenario design and deployment, was an even bigger collaboration project and took several months of work from multiple people, culminating in the challenge being deployed and played on in May 2024. This was an opportunity to showcase URSID's abilities on an even bigger scale, while again gathering valuable attacker data for research purposes. We will discuss the development and deployment of Casinolimit over this next section.

5.3 Casinolimit and BreizhCTF.

5.3.1 Overview and design constraints.

BreizhCTF *Breizh CTF*, URL: <https://www.breizhctf.com/> (visited on 07/10/2024) is a yearly Capture-The-Flag competition hosted in Rennes, with the 2024 edition taking place in May and having around 600 participants split into 120 teams. Several challenges are offered to the teams, which reward points whenever they manage to complete them - represented by them acquiring a flag, which is a sequence of characters found at the end of the scenario proving that they did complete it. Participants had 10 hours during the night to perform as many of these challenges as possible and gather points, before being ranked at the end based on their score. PIRAT^{13 14}, the research team of the author of this thesis, was offered to take part in it as an indirect consequence to the success of the CERBERE experiment, with URSID serving as the scenario design and architecture generation tool. However, while it still has values as an initiation to cyber-security, BreizhCTF is first and foremost a competition. Thus, scenario variability - which was at the center of CERBERE - is actually undesirable here, as having procedures differ between instances may lead to

13. <https://team.inria.fr/pirat/>

14. Formerly CIDRE <https://team.inria.fr/cidre/>

one group's challenge being unfairly easier than the other. Furthermore, such CTF events usually lead to participants posting write-ups of their endeavors at the end. In the event where procedural variation was involved, write-ups may differ from one participant to the other which would be confusing. Finally, a lack of procedural variation makes it easier to provide hints to participants if necessary.

Thus, while we believe URSID's procedural variation to be an important asset, the decision was quickly made not to rely on it for this particular event. Fortunately, thanks to our ability to describe scenarios on a procedural level directly, alongside our architectural constrain system, URSID is easily able to deploy scenarios with set procedures and guarantee that said procedures are compatible with each other, as long as those are implemented in the first place. We are also able to reuse procedures already implemented in CERBERE, highlighting the efficiency of URSID.

The skill level of participants was also expected to be higher than for CERBERE, with some of the top teams actually being part of professional pentester organizations. We also still had an incentive to make the event interesting and entertaining for the players. Indeed, this was one of the many challenges available to the participants, who were on a timer to complete as many of those as possible. Thus, a challenge being too dull or too difficult may result in the players quitting and trying another one. On the contrary, an engaging, entertaining and well calibrated challenge would potentially lead to more players attempting it and staying, providing us with more valuable attacker data and giving us better returns on our investment and development time¹⁵.

The scope of the event was much different than CERBERE, with much more expected participants: 600 split in 120 teams as opposed to a dozen. Hosting every generated URSID instance on a single machine was thus out of the question this time. We instead turned ourselves to cloud deployment solutions, landing on the French cloud provider OVH OVH, *op. cit.* in order to provide a wide array of servers able to host our virtual machines. This choice, as well as most if not all the implementation of the cloud deployment was handled by the same engineer who was in charge of the previous experiment's deployment. Casinolimit was overall an ever bigger collaboration than CERBERE was, with several researchers, interns and PhD students taking part in the design, development and deployment of the scenario. This is in part due to the much larger scope of the event: more instances, more budget, more potential data and the project being officially supported by inria making the stakes higher as a result.

15. As well as being better publicity for both the PIRAT team and Inria.

While this thesis will mostly focus on URSID itself and the author's implication within the project, we will specifically mention other team member's contributions when relevant. This project spanned six months, with regular meetings in order to discuss the scenario and its implementation. Having established the goals of this experiment as well as its specific challenges compared to CERBERE, we will first discuss the scenario we ended up designing, how this affected URSID's design, before going into the results and takeaways of the experiment.

5.3.2 Scenario design and implementation.

As mentioned in the previous section, signing a scenario to take part of a CTF event offers unique limitations. First, the attack scenario has to be difficult enough to offer a worthwhile challenge to the participants, but still has to be beatable within the given time frame. The scenario should ideally be appealing, entertaining and offering variations in the questions it asks the participants, despite the fact we are not able to rely on URSID's procedural variation this time. We are also limited by URSID's deployment abilities, i.e. Linux instances only.

For the variability part, we wanted to provide a variety of services, even greater than those offered in CERBERE. Offering some form of web challenge was an easy choice, especially since some of the same people who designed CERBERE's web-related procedures were involved in BreizhCTF. Databases are also a standard service that can be expected to be part of any serious organization. We came up with the idea of requiring the attacker to create an account on a website by using an email confirmation that they would receive on a previously compromised email account. This led to the development of an email server as a service. This also allowed us to increase the liveliness of our scenario, as we were now able to populate the server with emails that were either relevant to the attacker or just appropriate for the chosen setting. Attacker actions could also trigger emails to be sent, making the scenario more responsive to their actions and simulating real user activity. Finally, we came up with the idea of a remote camera that would have to be manipulated to reveal credentials written by hand on a real-world object. We thought the physical implications of this action would be particularly entertaining for participants.

Our final design includes four machines, a slight increase from the three available at CERBERE. We felt that this increase was reasonable, as participants were expected to be more skilled, work in teams, and have more time to complete the challenges. To increase the appeal and entertainment value of the scenario, we came up with a synopsis

for the challenge that would involve the compromise of a casino. Our first idea was that a customer of the casino would try to pay off his debts. For ethical reasons, this was changed to a former gambler who has paid her debt and is attempting to exercise her right to be forgotten by removing herself from the casino’s database. This former player, named Camille¹⁶, has paid her debts and has made several attempts to contact the casino to have her record removed, but has received no response. Participants act as Camille, who has obtained the login credentials of a casino security officer (named Casinolimit) and is attempting to remove her name from the casino’s main database.

Figure 5.8 showcases a graphical representation of the scenario we designed, developed and deployed, while figure 5.9 offers the corresponding scenario on a procedural level according to URSID formalism. Each instance is composed of 4 machines in which attackers are expected to perform 9 transitions spanning 7 MITRE ATT&CK techniques, which due to the lack of procedural variation translates to 7 procedures. Just as in CERBERE, each participant would first be given a URL corresponding to a website which was part of their specific instance. Unlike CERBERE however, the first machine would this time just be a starting point in which no exploits were expected to be performed, instead acting as a way for attackers to access the rest of the network. URLs were provided to participants through custom playing cards during the event, which contained the URL of their instance as well as an SSH login name. Accessing the URL would lead participants to a simple HTML page which described the goal of the challenge, some legal conditions and asked them to consent for their attacker data to be logged in. Upon accepting these conditions, attackers would be provided with a fixed password, which combined with their SSH login name would give them access to the first machine.

This experiment was by nature more ambitious than the previous one, which required several new additions to URSID on a technical standpoint. First, since this was a collaborative work between several researchers, we wished to make the integration of different code parts (websites, services...) easier than it was for CERBERE. Thus, some file constraints associated to these procedures now point to git repositories in order to fetch their files, as opposed to having all the required files downloaded locally. Secondly, information about each machine which could not be added through architectural constraints were sometime required. We identified two main categories of such information:

- Network related information, such as DNS servers, IP addresses and overall network structure (i.e. which machine belongs to which network). These points were not

16. Named for unisex reasons.

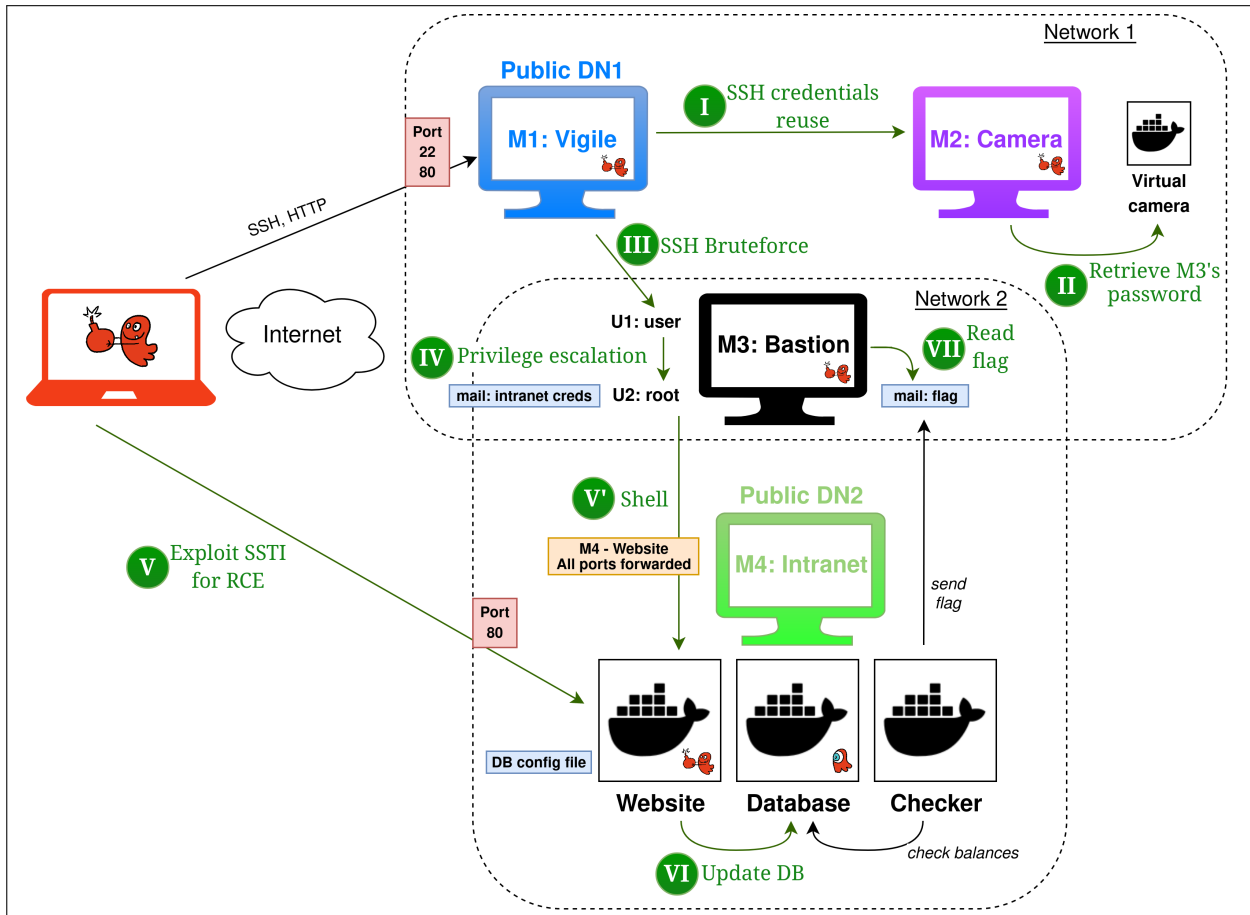


Figure 5.8 – Graphical representation of the Casinolimit scenario.

relevant for CERBERE as every machine could see each other and shared the same default network, but will be relevant here due to our desire to make this scenario more complex and realistic.

- Quality of life information related to the fact this is a CTF experiment, and as such has special needs of entertainment value and difficulty. These include hints (mostly provided in the form of customized login banners), software being installed on machines by default outside procedural constraints, additional users being added as ornaments and logging software being setup on the machines.

These have been added via an additional configuration yml configuration file that works in conjunction with the json scenario description. This file is able to customize the network configuration of machines by specifying IP address ranges for various services, as well as what network these machines belong to. These values are then reused by Ansible

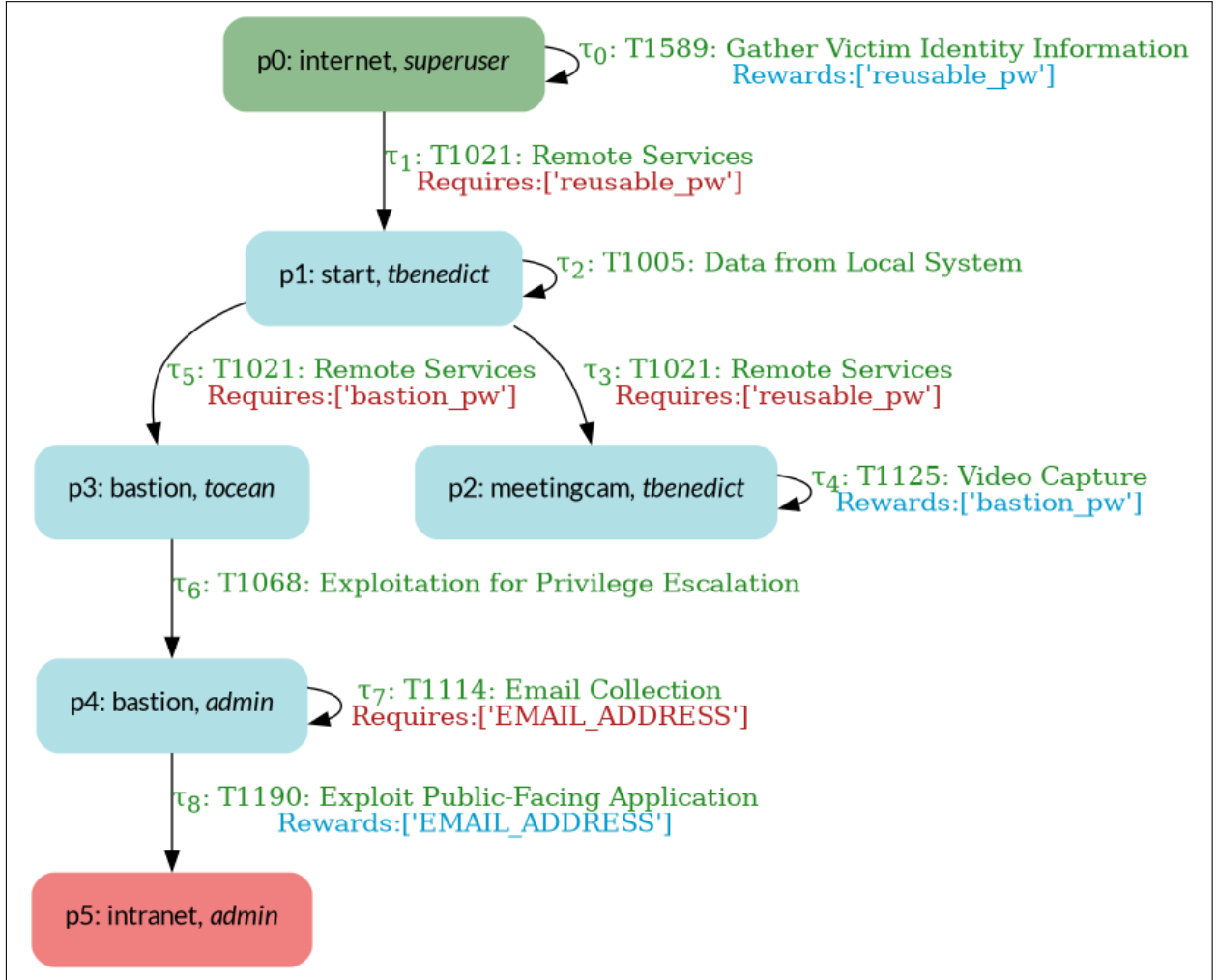


Figure 5.9 – Casinolimit scenario according to URSID formalism.

as variables to populate the configurations of those services. Other options-such as adding ornamental users or software-are internally associated with architectural constraints, which are then added at the beginning of the refinement process.

Figure 5.10 showcases the configuration file used for Casinolimit. For instance, machine *meetingcam* having the option **camera_banner** - which adds a SSH login banner relevant for this machine - pulls architectural constraints from a file named **camera_banner.json**, which contains file constraints adding that banner to the machine. The **global_network_pattern** entry sets the IP range of every machine of the scenario, and is useful within scripts in order to identify the IP of the machine within the network. Finally, machines may specify which network they belong to (such as *bastion* belonging to **network1** and **network2**),

whether they are a DNS server and for which network (*bastion* is one for `network2`), and if not they may point to a DNS server to use as their own - such as intranet. Finally, we may set the value of some secrets to hard coded value - such as `reusable_pw`, which will always be equal to `NEi9g8Bc`. This initializes the value of this secret within our global secret dictionary before the refinement process starts, guaranteeing that every procedure and transition related to this secret will use the same value. This is useful in the event where we want tighter control over secret values. In the case of Casinolimit, we wanted this secret value to be the same among all participants and match the one provided to them at the beginning of the experiment.

Such edits allow us to have more granular control over the machines while still respecting the URSID paradigm of constraints, since only procedures compatible with these options will be selected by our procedure refinement algorithm. We will mention the use of these additional configurations whenever relevant when describing the scenario. Note that each machine has `auditd` installed, which is necessary to construct our attacker dataset.

The outline of what attackers were expected to perform , and the corresponding procedures are as follows. As a reminder, due to the conditions of the experiment, there is no procedural variation involved this time, meaning each transition will be associated with a single procedure.

- First, participants are provided with the login and password necessary to access the first machine of the scenario through a SSH connection. This is represented by procedure π_1 : *Credentials physically provided to attackers*. We made this procedure part of technique T1589: *Gather Victim Identity Information*, as this is akin to an attacker performing investigative work prior to entering a system. We chose to represent this procedure as taking part entirely on the attacker's machine. As a result, it does not lead to any proper architectural constraints. However, this procedure still rewards a password secret, which will then be used in the rest of the scenario. We made this password identical for each generated instance, as a) this step is not meant to be difficult b) the website which provides the password can be static as a result.
- Once they acquire these credentials, attackers may perform procedure π_2 : *SSH from password* in order to access the attack position (*start, tbenedict*), with *tbenedict* being here the name of a fictional security agent whose account has been compromised by Camille. This procedure corresponds to technique T1021: *Remote Services*. This procedure was already implemented in URSID before as part of our APT-3 inspired scenario and was reusable immediately. It requires Linux, makes edits to an SSH

```
global_network_pattern: 10.*35.*.*
vms:
  bastion:
    options:
      - auditd
      - bastion_banner
      - dns_server network2
    networks:
      - network1
      - network2
  meetingcam:
    options:
      - camera_banner
      - auditd
      - sshpass
      - expect
    ornament_users:
      - nbreen
      - twiseau
      - cfragasso
    networks:
      - network1
  start:
    options:
      - auditd
      - start_banner
      - sshpass
      - expect
    networks:
      - network1
  secrets:
    reusable_pw:
      NEi9g8Bc
  intranet:
    options:
      - auditd
      - casinolimit_banner
    dns_server:
      10.135.*.20
    networks:
      - network2
```

Figure 5.10 – Additional configurations for Casinolimit scenario.

configuration file in order to allow password connections, and makes sure the password of the user of this attack position matches the one in the secret.

- π_3 : *Simple Local Website on port 80 (requires sudo privileges)*, which corresponds to T1005: *Data from Local System*, is how we implemented the landing website which provided participants with the information about the scenario. This website is launched as the root user as opposed to the one associated with that attack position in order to be able to put that website on port 80, as this action requires administrative privileges. While this may make this machine less secure in the long run, as launching services as a root user always do, this fact that this machine is only intended as a starting point for the scenario with no attached exploits makes it a non concern. One may argue that this procedure should have been the one rewarding the secret `reusable_pw` as opposed to π_1 , but we wanted a way to represent that attackers were provided with information about the scenario before landing on (*start*, *tbenedict*). This procedure requires Linux, python, pulls files from a git repository associated with the website, and starts a service as root. Files for the website were designed, generated and uploaded on this git repository by other members of the design team. A visual representation of this website is available in figure 5.11.
- Participants are then expected to go to machine *meetingcam* as the same user *tbenedict*, using π_4 : *SSH from password*, which is identical to π_2 . The fact users have to reuse the same password was hinted at by machine *start* login banner, which is added through additional options (visible in figure 5.10). This banner also hints them as to the nature of that machine (as it mentions a CCTV), as well as inciting them to check their emails on Bastion, which will be relevant later in the scenario. Theoretically these banners could have been part of the procedure itself, but that would have made them hard to reuse for other scenarios.
- Once on this machine, participants may perform π_5 : *Simulated camera with password*, which corresponds to technique T1125: *Video Capture*. In this procedure, attackers are expected to manipulate a simulated camera through bash commands, which let them move the camera up, down, left or right as well as take snapshots. This would let them acquire a password written on a whiteboard in the same room as the camera, with one letter missing which they were supposed to a brute-force attack. The white board also contains hints towards the rest of the scenario, with the name of a specific CVE being written. This procedure originally involved a real life camera, but concerns about several participants using the camera at the same time resulted

Propulsé par

Créé par

Incubé à

CentraleSupélec

Casino Limit

Faisons respecter le droit à l'oubli !

Dans cette épreuve, vous incarnez Camille qui a trop longtemps fréquenté Casino Limit. Après avoir perdu beaucoup d'argent, vous avez arrêté de jouer et payé votre dette. Pour tourner la page, vous avez envoyé un formulaire de *droit à l'oubli* auprès du casino qui ne l'a pas respecté.

Votre objectif :

Faire respecter votre droit à l'oubli en supprimant vos données de la base de données du casino !

Consentement à la collecte de données

Ce challenge vous est proposé par l'équipe [PIRAT \'\) ;](#) qui est une équipe de recherche conjointe à CentraleSupélec, au CNRS, à Inria, à l'Université de Rennes au sein de l'IRISA.

Pour nos travaux de recherche, nous analysons les procédures d'attaques dont celles des participants de CTF. Lors de cette épreuve, nous effectuons une **collecte de données** des commandes bash et du flux réseau sur les machines du challenge.

Figure 5.11 – Landing page for Casinolimit challenge, including information about the setting of the scenario and informing participants their actions will be recorded.

in each instance simulating its own camera instead through a docker container, developed by other members of the design team. This procedure pulls files from this git, has a docker service and modifies the `bash_aliases` file in order to provide attackers with the relevant commands. The photography taken and made available to attackers through the simulated camera is visible in figure 5.12. It also rewards them with a secret, which has a fixed value as only one password photography was available. Note that this machine also had a few additional users (cfragasso, nbreen, twiseau¹⁷), which were added through the use of our yml option file. These users

17. Named after infamous movie directors.

are only intended to increase the credibility of the scenario and be additional users of the casino’s CCTV, but access to these positions was not required to complete the scenario, hence why they are missing from the attack graph.

- Using this credential (which does require brute-forcing of one of the characters), participants may now move to $(bastion, tocean)$ through π_6 : *SSH from password*, which is identical to π_2 and π_4 . Thanks to URSID’s secret system, the password of this account will indeed match the one found with the camera which rewarded the `bastion_pw` secret. Participants may find this step with a combination of network analysis (nmap is made available for the entry position of this procedure in order to facilitate this process) and the hints provided by the white board and login banners.
- Once in this position, attackers may elevate their privileges by applying π_7 : *Vulnerable OverlayFS subsystem (CVE-2023-0386)*, belonging to technique T1068: *Exploitation for Privilege Escalation*. This will provide them with the new attack position $(bastion, admin)$. We could have reused one of the procedures available for this technique in CERBERE, but instead took this opportunity to enhance our catalog of procedures. This procedure, which requires a specific Linux kernel version in order to be performed NIST, *CVE-2023-0386 Detail*, 2023, URL: <https://nvd.nist.gov/vuln/detail/CVE-2023-0386> (visited on 04/20/2024), is related to a vulnerability discovered in 2023. While the technical details behind the exploit are non trivial, exploiting it requires a simple download and execution of a github script. This CVE was also hinted at through the whiteboard participants saw on the camera beforehand. This procedure is actually the first instance of an URSID procedure being implemented and added by someone else than this thesis author, showcasing the potential collaborative aspect of this tool. It requires a specific kernel version (installed through a custom script), Ubuntu 22.04, and the end user to have administrative rights on the machine.
- From this administrative position, attackers are expected to perform π_8 : *Casino Emails through postfix agent*, associated with technique T1114: *Email Collection*. Plenty of emails are available to the attackers, including some from Camille asking to be removed from the database, automated user creation related emails, as well as a complete pentest report of the casino’s intranet which will be relevant in the next steps. Some of these emails were generated by URSID itself alongside the generation of the machines, while others were dynamically sent through the casino’s intranet. Note that the scenario description mentions that this procedure requires a secret

named `EMAIL_ADDRESS`, which would be rewarded by the next transition. Indeed, attackers will then be expected to create an account on the casino's internal website by using the email box they just got access to. As the website needs to know the mail address of the mail server in order to send it emails, we chose to represent this through a secret in our scenario, as the way we handle these guarantee that this value will be consistent throughout the two machines. This procedure installs and configures the postfix package as its mail server, offers snail and mutt as mail clients to the attacker for easier configuration, and edits several configuration files. Note that this machine will also be a DNS server (necessary in order to be able to send email to it directly), as well as a bridge between the two networks making the scenario. Both of these factors are handled through our yml option files as opposed to procedures.

- From this position, attackers are now in the same network as the *intranet* machine¹⁸, providing them with access to the website hosted by position (*intranet, admin*). Attackers are expected to perform procedure π_9 : *Website for breizh CTF*, which is part of technique T1190: *Exploit Public Facing Application*. Just like for the first website in CERBERE, while this is technically a single procedure for attackers to exploit - as it corresponds to a single technique -, this part of the scenario actually requires the attacker to chain multiple exploits. Attackers must first create an account on the website, which represents the intranet of the casino. Using the sign up function offered on the landing page will send an email to the server hosted by *bastion*, hence the `EMAIL_ADDRESS` secret being rewarded. Once this account is created attackers may explore the intranet and find an exploit within the website. By combining CVE-2022-29078 and an insecure image upload function, as well as carefully crafting a URL address, attackers may execute any script through the host of the website. In the event where they execute a SQL script in order to interact with the account balance database - which is visible in the website itself -, attackers may remove Camille's entry from the database, accomplishing the goal of the scenario. The steps of this procedure - here heavily summarized - were hinted at through both comments left in the html code and the pentest report participants were provided with in an email hosted on *bastion*. Despite these, as we will see in the next section, this was arguably the hardest part of the scenario, requiring attackers to chain multiple exploits together with handcrafted scripts, as opposed to simply downloading an

18. not represented on the scenario description

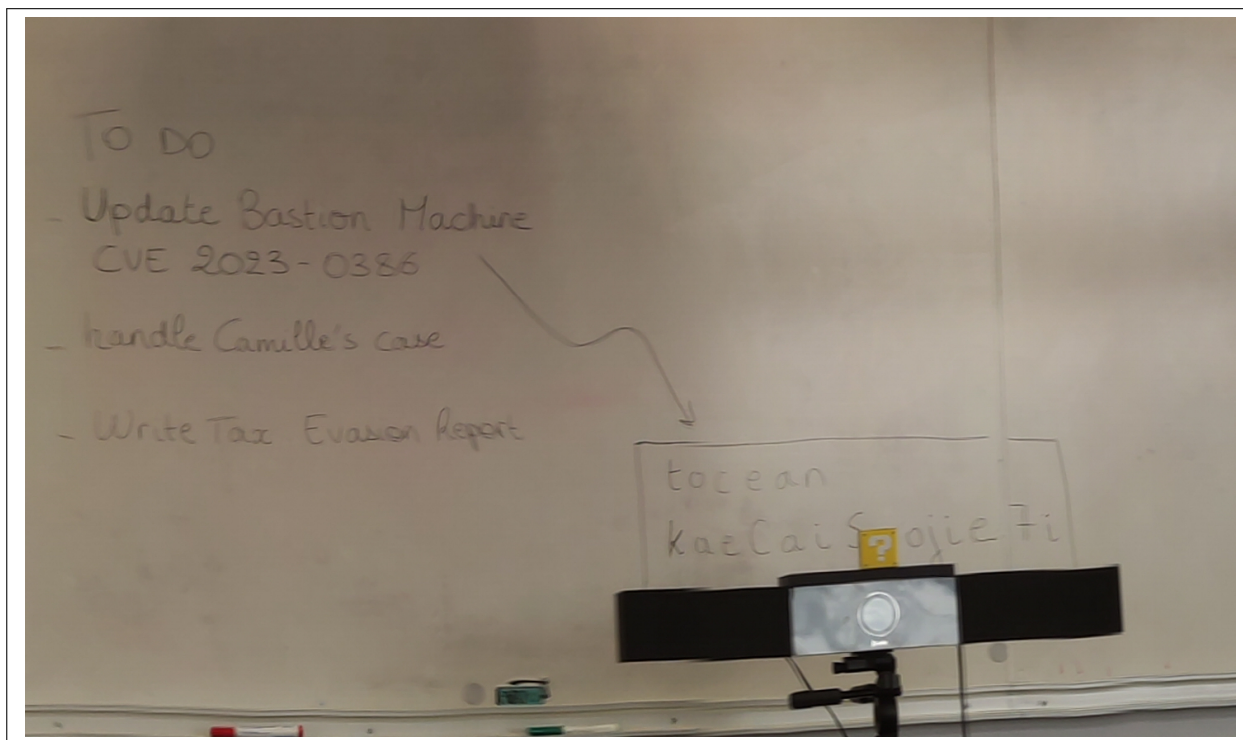


Figure 5.12 – Photography of credential (with one hidden character) available to attackers through the simulated camera.

exploit on github. This procedure pulls files from a git repository and launches the appropriate services. The website and associated database were developed by another researcher of the team, and provided through the form of a docker.

The attack position (*intranet, admin*) is shown as a winning node, as it would be theoretically possible to land on this position by performing a reverse shell using the previously mentioned exploit - which some attackers actually did! It is not strictly necessary to access this position to win this scenario however, as the goal of the challenge is an automated email sent to (*bastion, admin*) once Camille's record has been erased from the casino's database, which contains the flag necessary to score points.

All these procedures were implemented in URSID, which combined with the scenario description (on a procedural level) and designing the yml option files lead to the generation of Ansible playbooks. These playbooks could either be played locally for testing, or pulled as part of an automate production workflow leading to the deployment of these scenarios into OVH cloud machines.

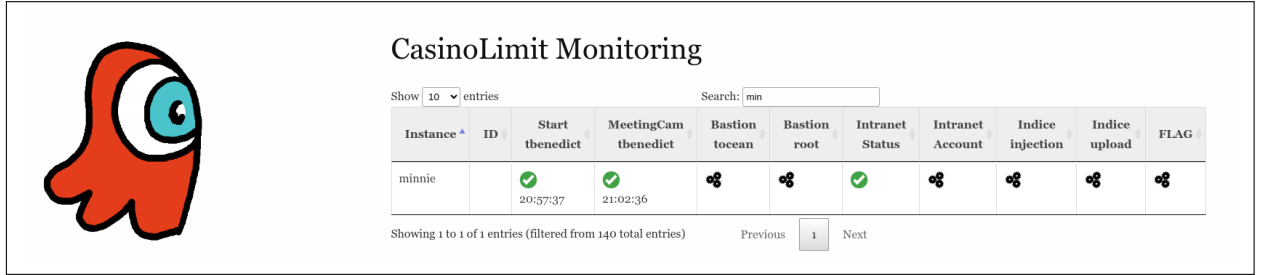


Figure 5.13 – View of live monitoring of one of the breizhCTF instances.

We mentioned in CERBERE that we wished to have the ability to monitor attacker movement within the scenario during the attack itself, as opposed to waiting until the end of the experiment. This was achieved for Casinolimit thanks to a more elaborate logging setup. Each instance had filebeat Elastic, *Filebeat*, URL: <https://www.elastic.co/beats/filebeat> (visited on 07/26/2024) installed on it in order to regularly send its system and auditd logs to centralized server. This server would then use logstash Elastic, *Logstash*, URL: <https://www.elastic.co/logstash> (visited on 07/26/2024), a data processing pipeline software, in order to parse and organize these logs. Finally, these logs would be automatically parsed every 2 minutes for patterns of compromise and attacker advancement, which would then be sent to a visual dashboard accessible to our team in real time. Figure 5.13 offers a view of this dashboard. This implementation was mostly handled by other members of the development team, but URSID did help in automatizing and installing filebeat on each of the instances.

The scenario was regularly deployed and tested with every new iteration of the implementation. In order to help with this task, a prototype of an automated attacker agent able to perform parts of the scenario on its own was created by one of the members of the team. While this thesis author had no involvement with the code of that agent, it did help to detect issues with new versions of Casinolimit. Furthermore, the idea of generating automated attackers alongside scenarios - just as we wish to generated automated log analysis patterns associated with a given choice of procedures - is one we will keep in the future.

Reflections on URSID’s use as a CTF generation platform We made significant improvements on URSID between the CERBERE experiment and the generation and design of Casinolimit, and in particular its use as a collaborative design tool. Several

researchers of the team were able to make edits on URSID's procedures themselves, either by tweaking existing files, working on separate repositories which were then pulled as part of a procedure (for instance for website implementations), and one of them even implemented their own procedure from scratch. However, while there were communications between our development team and the organization responsible for the CTF event - who for instance had to be provided with the flag we chose as a reward for completing the scenario -, the organizers had no involvement with the use of URSID itself. Scenario design, implementation and deployment were all handled by our development team, who all had prior experience with the tool as well as direct contact with its main developer¹⁹. Thus, URSID's ease of use by organizations unfamiliar with the tool was not directly evaluated for this experiment.

Nonetheless, our internal development process highlighted such potential problems for future widespread use of the tool. In particular, our previous implementation of networking had to be significantly reworked for this experiment and, due to time constraints, was focused on making this specific CTF event work. While this work will be reused as the basis for future networking implementations, we found that improvements to its flexibility would be desirable for a better user experience. Our deployment workflow was also improved from CERBERE thanks to an automated Gitlab deployment pipeline that took the default output of URSID and used it as a basis to generate machines on our OVH instances. However, this pipeline was manually created for our specific needs for this experiment and is not part of the tool itself. Testing the Casinolimit implementations was done by deploying instances on a local network through URSID's native pipeline before testing on our remote instances through this custom pipeline. Ideally, the tool itself would be able to deploy scenarios based on different network paradigms and architectures. Casinolimit helped us identify these two main areas of improvement for URSID, which will be the focus of our future implementation work for the platform.

5.3.3 Event and results.

The event started on May 17th from 20h30 until 7h the next day, with an expected turnout of around 600 participants divided into 120 teams. Three members of our Casinolimit design group were physically present during the event to oversee our deployments, provide participants with their instances via physical playing cards containing the required

19. The author of this thesis.

URL and login, and answer questions and provide occasional tips via a message board. These hints were intended to point attackers in the right direction if they got stuck and asked for help, and to ensure that their problems were not related to an instance failure on our end, in which case they would be provided with a new scenario deployment. This happened a handful of times throughout the night, as running scripts on the website hosted by (*intranet*, *admin*) could, depending on the content of the script, cause that service to be shut down, making the website inaccessible and the scenario unbeatable. Clues were also intended to help contestants, but out of concern for the integrity of the competition, they never directly provided the answer, but rather nudged contestants in the right direction if they were lost or chasing dead ends. For example, several attackers failed to read the pentest report that was sent to them as an email attachment in the *bastion* machine, and were directed to look at it in more detail as it contained relevant information about the exploits to be run on the website hosted by *intranet*.

Figure 5.14 shows the results of the experiment as detected by our automated logging system. Around 110 participants reached the (*start*, *tbenedict*) position²⁰, which was trivially accessible as the password was directly provided by the starting website. These numbers drop to 50 for position (*meetingcam*, *tbenedict*), which while not a hard transition to take did require participants to immerse themselves within the setting of the scenario - i.e. perform a network scan and read the login banner. We may thus consider that this is the number of participants who seriously attempted the scenario, as opposed to the 110 who most likely checked out the setting but decided against pursuing the scenario. This drop off may be due to time constraints as well as competition from the other challenges of the BreizhCTF event, as time invested in our Casinolimit scenario was done at the cost of spending time on other instances, which may be more attractive or be perceived as a higher return on investment in terms of points per minute. In absence of numbers from other challenges however, it is difficult to say whether our drop-off ratio was high, normal, or lower than the average challenge.

From these 50 participants who managed to reach (*meetingcam*, *tbenedict*), one can see that almost all of them managed to create an account on the intranet of the casino. They managed to gather the credential using the simulated camera, performed a brute-forcing attack in order to guess the missing character, landed on (*bastion*, *tocean*), performed a privilege escalation using a CVE (which was hinted at with the camera), found the mail server and the website and combined the two in order to acquire an account. We

20. Almost one for every team present during BreizhCTF.

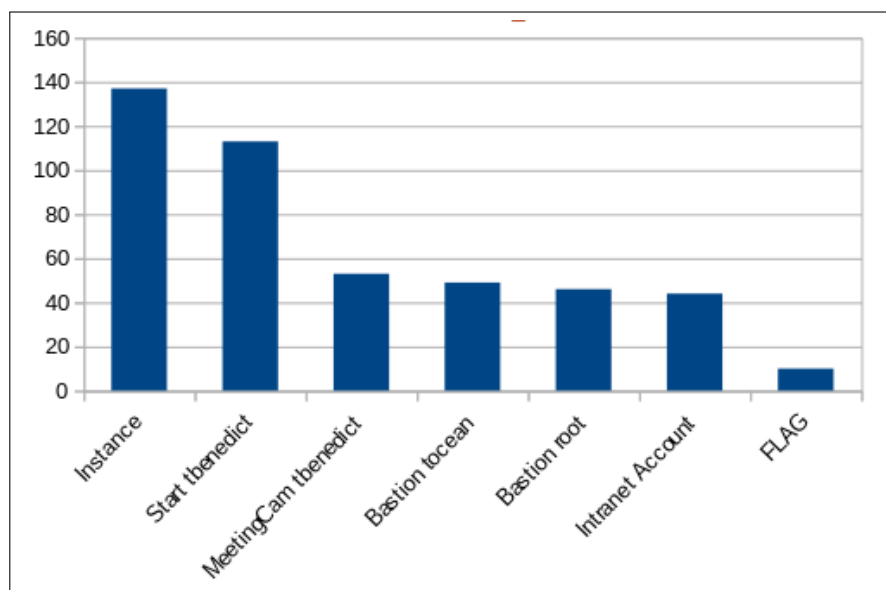


Figure 5.14 – Results of Casinolimit.

may conclude that these steps of the scenario, while nontrivial, were doable in the time provided to the participants, as about 90% (44) of the ones we consider having seriously tried the scenario landed in this position. This may be due to a combination of single step procedures and the hints made available throughout the scenario, with items such as the emails, login banners and the white board seen by the camera hinting to future steps.

However, the final step proved to be a significant challenge as only 9 teams managed to capture the flag. Discussions with participants during the event revealed that some spent several hours on this part of the challenge without success, and that the amount of points offered by the challenge may not have been worth the time invested. Nonetheless, the scenario was apparently beatable, and some of the participants even beat it without needing any hints from the team members present during the challenge. We can attribute these results to a) the difficulty of this part of the challenge being high, as it required chaining several custom exploits together, and b) this difficulty being significantly higher than the rest of the challenge, which may have enticed less skilled attackers to spend significant time on this challenge once they had beaten the easier parts, as a sort of sunk cost fallacy. However, this was very beneficial to our data collection goals for this experiment, and the increasing difficulty curve was intended for this reason. This is in contrast to CERBERE, where the hardest part of the scenario (another website) was at the very beginning, thus limiting the amount of logs we could harvest for our dataset.

Despite these occasional complaints about difficulty, anecdotally, based on discussions during the challenge, participants liked the challenge, with specific praise for both the camera procedure and the overall lore of the scenario.

This experiment generated 38 GB of system logs and 535 GB of network logs, which are now being used by several members of the design team for their own research. Aside from a few hiccups, such as the gitlab token leak and monitoring individual instances that became unavailable in the middle of the night, the challenge was overall a technical success, thanks to the hard work of the entire team over these months. Furthermore, parts of this work can be reused for future events, as it led to significant improvements in both URSID and the team's ability to deploy cloud experiments at scale. From a player's point of view, the main criticism was that the difficulty of the challenge was too high for the amount of points it rewarded, mostly due to the very last part - the website exploitation - being significantly harder than the rest. Future challenges may take this difficulty curve into account more, or reward points for completing individual parts of the challenge - for example, rewarding a flag for accessing the mail server at all.

From URSID's point of view, we argue that Casinolimit was both a good showcase of its perks as a vulnerable architecture deployment infrastructure, as well as an excellent opportunity to improve on the design and technical abilities. It did however highlight some weaknesses in its current implementation - particularly in the networking aspect -, which while were successfully solved for this specific experiment were identified as warranting improvement in the future. We identify the main takeaways of this challenge on both these aspects as follows:

- URSID managed to scale up to a significantly bigger deployment than CERBERE, with 600 virtual machines being deployed as opposed to 45. While this deployment part isn't part of URSID's core yet and was the result of manual implementation work from a dedicated engineer, this work still makes use of URSID's output Ansible playbooks in order to provide configurations and procedures to instances. Due to development time constraints, our way of freeing URSID's output from its Vagrant and VirtualBox dependencies was not the most technically sound, but this experiment showcased it was possible. Improving on this aspect - i.e. making it easier for a scenario designer to choose its deployment setup, whether it is a local VirtualBox generation or an Openstack cloud large scale operation - is one of our current development priorities.
- Collaborative work was improved from CERBERE, with multiple git repositories

being developed by other members of the team and pulled in URSID during generation directly.

- Logging was improved, with live monitoring of instances being made available alongside automated log gathering for post-experimental analysis.
- Despite being designed with procedural refinement and technical level scenarios in mind, challenges required fixed procedures (such as a competition like BreizhCTF) are still fit for URSID to deploy, thanks to its abilities to define scenarios on a procedural level and the flexibility of our architectural constraint system.
- Previous work from CERBERE and the APT-3 inspired scenario was able to be reused, with the *SSH from password* procedure requiring no additional development time to be added back in Casinolimit. This showcases that while there is a technical cost to adding new procedures to URSID, this work is a one time investment. Improving URSID's procedure catalog is another of our main deployment goals for the future.
- The secret system is flexible and efficient at guaranteeing text entries to share the same values across the network. While it already showcased these properties for traditional credentials, such as passwords or SSH keys, we here made use of it in order to share an email address value between machines *bastion* and *intranet*.
- Networking configurations took a lot of development time, and our current fixes (relying on additional configuration files indicating IP ranges for several services) may not scale well to other future experiments. This is a combination of a lack of networking representation on technical level scenarios, code reliance on VirtualBox to assign IPs locally - requiring us to ignore that output for other deployment setups - and overall lack of planning of networking features when originally developing Apiculteur, URSID's module for converting procedural refinement output into Ansible playbooks. This is not to diminish the development work accomplished for URSID thus far, as we had significant deadlines to meet for both CERBERE and BreizhCTF, which alongside limited development resources lead to a "get it to work first make it flexible later" approach. Working on BreizhCTF both highlighted URSID's weaknesses in this aspect, but also provided us with useful insights and practice in how networking conditions may be implemented in the first place.
- Scenario formalism is not perfectly accurate to an attacker's actual path within the scenario, with simplifications in the later parts (particularly the email server and

website) being made in order to fit our deployment setup at the time. Representing complex actions - such as an attacker using a website and mail server in tandem as is required for the last part of the scenario - are difficult to accurately represent on such a high level of description. Uses of secrets for transitions is however a successful way of representing attacker required movement within a scenario.

- Most procedures implemented within this scenario - such as the *intranet* website or the camera - are specific to this particular setting and should arguably not be reused for other scenarios, as they require knowledge of the overall scenario lore (the casino, the initial website, the card being physically provided) in order to make sense. While their specific requirements and rewards make them unfit for most natural procedural selection so far thanks to our procedural refinement algorithm, they - and other procedures specific to themed experiments - should ideally be excluded from the process altogether.

5.4 Conclusion.

Both experiments successfully demonstrated the capabilities of URSID in deploying vulnerable architectures, while providing a clear goal to motivate its development. CERBERE, an experiment with clear educational goals in mind, successfully used the procedure refinement process to provide participants with variability within their instances, while only requiring the scenario itself to be designed at a high level. The addition of new procedures-such as different privilege escalation exploits or variations of the service to be exploited as an entry point-was facilitated by this technique-procedure paradigm, which we argue is URSID's greatest strength over the rest of the state of the art. Casinolimit, on the other hand, was more of a test of URSID's ability to scale its deployment and be part of a professional capture-the-flag event, played by over 100 different teams over several hours and involving the collaboration of several researchers and developers over 6 months.

In both cases, URSID demonstrated that its ability to natively deploy local instances could be scaled up for more ambitious setups, albeit with manual effort, and could adapt to different deployment infrastructures. It also successfully collected attacker data in both experiments, leading in the case of CERBERE to a publication Besson et al., « CERBERE: Cybersecurity Exercise for Red and Blue team Entertainment, REproducibility » and the dataset being made available online to the research community. Such datasets are very valuable for applications such as machine learning or training automated data analysis

software, but can be difficult to obtain. URSID's ability to provide instances set up with logging software that can be played back can be of great use for such purposes, and has been of great interest to our research team. In addition, experiments conducted with this tool can be re-deployed to replay the scenario or generate more data for personal use. While such deployments would not share the same network setup as CERBERE and Casinolimit, but would instead be deployed on local networks on one's own machine, this may be of value if one wishes to replay the scenario for educational, entertainment, or additional data generation purposes. CERBERE was made available in URSID itself BESSON, *URSID*, along with documentation on how to use it. Casinolimit is also available through this repository, but an additional repository containing only the files relevant to this scenario was also created and made publicly available Sanchez, *op. cit.*, as several BreizhCTF participants expressed interest in retrying to beat the challenge. Both of these experiments also demonstrated that once a procedure is implemented, perhaps with a specific scenario in mind, it is much easier to use it as part of any other scenario. Researchers are then free to create technical-level scenarios using techniques for which procedures have been implemented, or to choose their procedures directly for some transitions. Thus, a growing library of procedures could lead to a growing library of scenarios using those procedures, with minimal implementation costs once the initial work is done each time.

However, these experiments helped us to identify several key areas where URSID could be improved, particularly in terms of its networking capabilities. In addition, we believe the tool has room to grow in terms of implementation flexibility and ease of adding new procedures. Some team members successfully added their own procedures during BreizhCTF - such as privilege escalation - but this required modifying some parts of Apiculteur's code itself, which is an unreasonable barrier to entry for a collaborative tool intended to facilitate such deployments. Overall, both experiments were successful proofs of concept for URSID, which now needs polishing to be a successful collaborative tool. However, it has clearly demonstrated its capabilities for the purposes of our research team, who are now familiar with how it works and have several plans for it that we will detail in our future work, ranging from using it to collect automated attacker datasets to other future live educational experiments.

CONCLUSION

6.1 Conclusion

The stated goal of this work was to provide a novel way of describing and deploying vulnerable architectures, with particular emphasis on the flexibility and reusability of a given description. This task is relevant for defense purposes as a way to train friendly cyber attackers (pentesters) to test and evaluate the security of existing networks, or to increase the overall awareness of the modus operandi of malicious actors in order to better combat them. In addition, controlled deployments of vulnerable instances are an excellent way to generate and collect attacker data logs, which are hard to come by and useful for various cyber defense purposes, such as training machine learning models to help security analysts detect malicious actions.

Thus, we wanted to make it easier to deploy variations of a vulnerable architecture that correspond to a single description of an attack scenario. We achieved this by focusing on two levels of description: a high-level based on ATT&CK techniques, generic enough not to be tied to specific architectural choices, and a low-level based on procedures. This dichotomy is our solution to the problem we identified in the state of the art of vulnerable architecture and cyber range generation platforms, which is their lack of flexibility and how designing new scenarios requires writing new configuration, as well as a lack of reusability of any given description.

First, we described our formalism for describing attack scenarios in section 2. By combining ATT&CK techniques with our concept of attack positions, which represent actual attacker sessions opened within the network, we are able to accurately represent a vulnerable architecture and possible attack paths within that architecture at a high level, thus freeing scenario designers from having to specify configurations for each machine. In particular, we offered the concept of attack positions - corresponding to login sessions opened by attackers within a machine - which, combined with the notion of secrets - data and knowledge acquired by the attacker during his journey - provide a way to accurately

represent the path of a malicious actor within an architecture. This high-level representation also frees scenario designers from being constrained by implementation details - such as operating system or software choices - in representing their desired challenge. However, such a description is not sufficient to be used as the basis for virtual machine deployment software as is, because decisions must be made about these implementation details.

We thus provide a way to convert this high-level representation into multiple low-level procedural descriptions, thanks to our procedural refinement process described in section 3. By choosing a procedure (a possible implementation of a given technique) for each transition within the scenario, and making sure that this choice does not lead to incompatibilities, we are able to propose several low-level implementations for a given technical-level scenario. These descriptions contain enough information to be converted and then deployed as vulnerable virtual architectures, a process we described in section 4. All of these descriptions and deployments correspond to the same original scenario, while potentially differing in their procedures. Attackers are offered the same overall path through each of these variations, but may encounter different challenges along the way. Because all of these variations are derived from a single (high-level) scenario description, this process provides a combination of abstraction (for the scenario designer) and replayability (for the attacker) that we believe is valuable in the context of cyber range and vulnerable architecture deployment. We implemented this two-level scenario formalism, as well as the notion of architectural constraints and procedural refinement, in an open-source tool called URSID, which was first used to represent and deploy a scenario inspired by the modus operandi of a real-world attacker known as APT-3.

Finally, we presented two successful experiments that used URSID as a backbone for scenario design and deployment. The first was CERBERE, which took advantage of the procedural variations offered by the refinement process to provide different experiences for each of the participants. Then, Casinolimit, a challenge designed as part of the BreizhCTF competition, demonstrated URSID's ability to scale to a larger deployment setup and provide a variety of services to participate in a scenario, including a camera, a website, an email server, and a DNS. Both experiments demonstrated the educational value of URSID, as well as its research purposes, thanks to its ability to collect system, network, and web attack logs corresponding to a designed scenario. Such datasets can then be used for various purposes relevant to researchers and defenders, such as automated blue team analysis and defense recommendations, or training machine learning models.

We believe this thesis (and by extension URSID's) main interests for the research

community to be as follows:

- The main work required from scenario designers is the implementation of new procedures if desired, which can admittedly be time-consuming. However, once this work has been done once, any procedure - as well as any techniques associated with those procedures - may be reused in other scenarios with little to no cost.
- URSID is designed with the ATT&CK matrix in mind, a widely used attacker classification nomenclature used by several other projects.
- Scenario descriptions are simple and easy to model and reuse.
- The constraint system is flexible and can describe most, if not all procedures.
- URSID is open source, documented and is in particular able to easily redeploy previously ran experiments (CERBERE and Casinolimit) as local instances.

6.2 Perspectives

The future work we consider for this thesis falls into two main categories. First, we consider planned improvements to the tool itself, motivated by the significant interest it has received from our local research team as well as some external stakeholders. While these improvements are more of an engineering problem than a pure research endeavor, solving these problems can significantly improve URSID's value as a research tool.

Indeed, the current implementation has its shortcomings, which may need to be addressed before URSID can be widely used by the research community. First, we encountered technical limitations during our two experiments, particularly due to our initial lack of network descriptions. This was due to a combination of time and manpower constraints, as most of the technical advancements were driven by the need to get both experiments up and running by the deadline, while being largely developed by this thesis author. This approach had advantages, especially as the needs and design of the tool evolved throughout the duration of this thesis, but has the disadvantage that the current implementation lacks maturity. A refactoring of the tool, especially in how it handles networking and deployment, would be desirable. Ideally, URSID would be inherently compatible with multiple virtual machine deployment paradigms, such as Vagrant, OpenStack, Proxmox, or even the Airbus Cyber range. Resources (in terms of time and manpower) are already being allocated internally to achieve this goal. We believe that there is still significant room for URSID to grow as a tool. This is because URSID has attracted a significant amount of interest from

various sources, motivating us to continue developing and supporting the tool and its ideas even after the completion of this thesis. As demonstrated by CERBERE and Casinolimit, its ability to generate attacker datasets is particularly relevant for the research purposes of several members of the PIRAT team. Its pedagogical values result in a local lab being successfully deployed and used for teaching purposes, as well as attracting interest from outside researchers with teaching responsibilities. The cyber deployment aspect has led to discussions with local companies as well as the French DGA¹, and honeypot applications are also being explored. While significant technical work may be required to get URSID to an acceptable state for all of these purposes and organizations, we believe it has potential for all of these applications.

URSID also already spurred interest from this thesis author's research team, which due to proximity and familiarity with previous experiments based on the tool are more able to make use of it immediately. The following list details future plans for the tool and project, ranging from experiments already on their way to long term development goals:

- Implementing and automating more advanced log gathering infrastructure as part of URSID itself, including the deployment of additional machines dedicated to receiving data from the scenario. The implementation of this functionality is already completed and going through testing as of the writing of this chapter, thanks to the work of an intern hired in part to improve on URSID's code base.
- Adding better network support by optionally representing it on the technical level description and taking it into account in the refinement and deployment phases. This work is also an ongoing work by the same team member, with some occasional input from this thesis' author relating implementation issues.
- Using URSID as a basis for honeypot deployment, which are vulnerable architectures intended to be exploited by outside attackers in order to gather data. This was the original goal of this thesis² and we believe URSID would be a valuable tool in order to deploy high-fidelity honeypots with varied entry points. As of the writing of this chapter this is an ongoing work by several members of the team, who successfully combined URSID and other vulnerable architecture deployment techniques in order to deploy and analyze several honeypots, with more complex experiments on the way.
- Overall refactoring of URSID's code, with the main goal being to make it easier

1. French military organization with a cyber division, original funders of this thesis.

2. An extended state of the art on honeypots did not make it to the final manuscript.

for outside users to add their own procedures and use it on their own networking environments without having to tweak the software itself. In particular, we wish to make URSID generation natively available on several deployment stacks such as OpenStack (used for Casinolimit but not automated by the tool itself) or customized IP ranges, as opposed to relying on local VirtualBox deployments. Several team members - including the Inria engineer who was in charge of deployment for both CERBERE and Casinolimit and is as such very familiar with the tool - are working on this.

- Improving on the automated attacker aspect, which would be able to test and exploit a designed infrastructure based on its procedural representation, by associating each procedure with actions to be performed. This would be useful for testing purposes, quick dataset generation and as a way to offer a solution for a given scenario to attackers for educational purposes. A working prototype based on this idea was developed for CasinoLimit, and there are talks to integrate it into URSID itself.
- Adding Windows support, in particular Active Directory deployment. Windows has been mostly ignored throughout this thesis due to engineering constraints and our choice to the (much easier to work with) Linux Operating System as a proof of concept for our deployments, but the dominance of Windows in IT fields and particularly in corporate environments would make it an extremely valuable asset to be able to emulate. This would require significant engineering work, as while we believe URSID's architectural constraint system to be flexible enough to handle Windows deployments, the process of converting these into virtual machines configuration files would be significantly different. This is a longer term goal compared to the previously mentioned ideas due to the perceived high engineering cost, but is high on our list of priorities nonetheless.
- Combining URSID with automated architecture analysis software (such as Bloodhound *BloodHound*, URL: <https://github.com/BloodHoundAD/BloodHound> (visited on 10/02/2023) or our own tool AWARE Manuel Poisson et al., « Unveiling stealth attack paths in Windows Environments using AWARE », in: *CSNet 2023 - 7th Cyber Security in Networking Conference*, IEEE ComSoc, Montreal, Canada, Oct. 2023, pp. 1–7, URL: <https://inria.hal.science/hal-04163780>), in order to generate attack scenarios corresponding to a given existing architecture in order to facilitate the work of pentesters. This is a long term project which would first require URSID to handle Windows and Active Directories deployments, but would be of

huge research interest.

Achieving some, if not all, of these goals would make URSID an even greater asset for research and education. Our ultimate goal for the project is for it to be open source software with an active community adding and sharing their own procedures for educational, data collection, and cybersecurity purposes, as well as designing their own scenarios. These scenarios would then be easily adapted and deployed on anyone's local network architecture, providing users with cybersecurity challenges to deploy, attack, and acquire data. There is a lot of work to be done to achieve this goal, but the internal use, support, and development of the tool, as well as several instances of external interest, make us optimistic about this future.

On the other hand, this thesis has laid the groundwork for additional research questions to be answered. First, we can explore the modelling itself and whether future implementation plans can be solved on a formalism level instead. For example, can we represent networking and service access on a high-level description while preserving the URSID deployment paradigm? Are our architectural constraints capable of representing Windows procedures as they are? Answering these questions would make it easier to add procedures, which in turn could lead to the prospect of automating this process. For example, one can imagine converting new CVE descriptions published by official organizations - usually with information about which software or operating systems are vulnerable to that CVE - into procedures represented by architectural constraints. Similarly, reports on specific attacker modus operandi are often published by multiple organizations, and machine learning solutions may be able to transform them into sequences of techniques and procedures.

We would then like to better evaluate URSID's capabilities as a learning platform through further field experiments, and use the results of these experiments to iteratively improve it. The variability aspect of deployed instances may be particularly relevant here: we could deploy multiple instances of a given scenario and ask for feedback on which ones participants found more instructive. Combined with the integration of learning elements into the tool itself - e.g. by associating procedures with hints on how to exploit them - one could imagine running a whole practical cybersecurity module at a university.

Finally, as mentioned above, we have plans to use URSID as a honeypot deployment platform, which would help us quantify how attackers in the wild respond to challenges of varying difficulty. Honeypot entry point variability is a particularly interesting area that is currently relatively unexplored, with most of the literature relying on remote access with weak or default passwords as a way to get automated attackers into their tool. We can

use this work to compare the efficiency and results of existing tools with more complex architectures, tweaking the difficulty of the entry point as well as providing lateralization options within the network for attackers once they are inside and monitoring their progress. Deployed architectures could be evaluated in terms of attacker interest and retention, matching them with the expected complexity, content and entry points (password, CVE, payload detonation...) of the honeypot.

BIBLIOGRAPHY

- Airbus, *CyberRange*, 2018, URL: <https://www.cyber.airbus.com/cyberrange/> (visited on 01/16/2024).
- al., Pham et, *CyRIS*, 2016, URL: <https://github.com/crond-jaist/cyris>.
- al., Uetz et, *SOCBED*, 2021, URL: <https://github.com/fkie-cad/socbed>.
- Ammann, P. et al., « A host-based approach to network attack chaining analysis », in: *21st Annual Computer Security Applications Conference (ACSAC 05)*, 2005, 10 pp.–84, DOI: 10.1109/CSAC.2005.6.
- Ansible, *Roles*, URL: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_reuse_roles.html (visited on 05/30/2024).
- aptnotes, *APTnotes data*, URL: <https://github.com/aptnotes/data>.
- Baillie, Callum et al., *CybORG: An Autonomous Cyber Operations Research Gym*, 2020, arXiv: 2002.10667 [cs.CR].
- Barrett, Clark et al., « Satisfiability Modulo Theories », in: *Handbook of Satisfiability, Second Edition*, ed. by Armin Biere et al., vol. 336, Frontiers in Artificial Intelligence and Applications, IOS Press, Feb. 2021, chap. 33, pp. 825–885, URL: <http://theory.stanford.edu/~barrett/pubs/BSST21.pdf>.
- Barron, Timothy and Nick Nikiforakis, « Picky Attackers: Quantifying the Role of System Properties on Intruder Behavior », in: *Proceedings of the 33rd Annual Computer Security Applications Conference, ACSAC '17*, Orlando, FL, USA: Association for Computing Machinery, 2017, pp. 387–398, ISBN: 9781450353458, DOI: 10.1145/3134600.3134614, URL: <https://doi.org/10.1145/3134600.3134614>.
- Berady, Aimad, « Comprendre les menaces sophistiquées : Intentions, moyens, manières et connaissances convergentes des adversaires », Theses, CentraleSupélec, Nov. 2022, URL: <https://theses.hal.science/tel-03861429>.
- Berady, Aimad et al., « PWNJUTSU: A Dataset and a Semantics-Driven Approach to Retrace Attack Campaigns », in: *IEEE Transactions on Network and Service Management*, Special Issue on Recent Advances in Network Security Management 19.4 (2022), pp. 5252–5264, DOI: 10.1109/TNSM.2022.3183476, URL: <https://inria.hal.science/hal-03694719>.

-
- BESSON, Pierre-Victor, *URSID*, URL: <https://gitlab.inria.fr/pirat-public/ursid> (visited on 05/30/2024).
- *URSID documentation*, URL: <https://ursid.readthedocs.io/> (visited on 05/30/2024).
- Besson, Pierre-Victor et al., URL: <https://gitlab.inria.fr/cidre-public/cerbere-dataset/> (visited on 07/05/2024).
- Besson, Pierre-Victor et al., « CERBERE: Cybersecurity Exercise for Red and Blue team Entertainment, REproducibility », in: *2023 IEEE International Conference on Big Data (BigData)*, 2023, pp. 2980–2988, DOI: 10.1109/BigData59044.2023.10386953.
- BloodHound*, URL: <https://github.com/BloodHoundAD/BloodHound> (visited on 10/02/2023).
- Breizh CTF*, URL: <https://www.breizhctf.com/> (visited on 07/10/2024).
- Brisse, Romain, « Exploration recommendations for the investigation of security incidents », Theses, CentraleSupélec, Feb. 2024, URL: <https://theses.hal.science/tel-04594042>.
- Chainalysis, *Ransomware Payments Exceed 1 Billion in 2023, Hitting Record High After 2022 Decline*, URL: <https://www.chainalysis.com/blog/ransomware-2024/> (visited on 07/26/2024).
- CISA, *Widespread IT Outage Due to CrowdStrike Update*, URL: <https://www.cisa.gov/news-events/alerts/2024/07/19/widespread-it-outage-due-crowdstrike-update> (visited on 07/26/2024).
- Costa, Gabriele, Enrico Russo, and Alessandro Armando, « Automating the Generation of Cyber Range Virtual Scenarios with VSDL », in: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* 13.4 (Dec. 2022), pp. 61–80, DOI: 10.58346/jowua.2022.i4.004, URL: <http://dx.doi.org/10.58346/JOWUA.2022.I4.004>.
- Cyber Security, Centre for, *SolarWinds: State-sponsored global software supply chain attack*, URL: <https://www.cfcs.dk/globalassets/cfcs/dokumenter/rapporter/en/CFCS-solarwinds-report-EN.pdf> (visited on 07/26/2024).
- Elastic, *Filebeat*, URL: <https://www.elastic.co/beats/filebeat> (visited on 07/26/2024).
- *Logstash*, URL: <https://www.elastic.co/logstash> (visited on 07/26/2024).
- Ferguson-Walter, Kimberly, « An Empirical Assessment of the Effectiveness of Deception for An Empirical Assessment of the Effectiveness of Deception for Cyber Defense Cyber Defense », PhD thesis, Feb. 2020, DOI: 10.7275/z0rb-ek46.
- Ferguson-Walter, Kimberly et al., « The Tularosa Study: An Experimental Design and Implementation to Quantify the Effectiveness of Cyber Deception », in: Jan. 2019.

-
- FireEye, *Annual FireEye Mandiant M-Trends Report Reveals Global Statistics and Insights from Hundreds of Diverse Intrusions*, URL: <https://www.mandiant.com/company/press-releases/annual-fireeye-mandiant-m-trends-report-reveals-global-statistics-and-insights-hundreds-diverse-intrusions> (visited on 07/26/2024).
- Hashicorp, *Packer*, 2013, URL: <https://www.packer.io/> (visited on 07/02/2023).
- *Vagrant*, 2010, URL: <https://www.vagrantup.com/> (visited on 10/23/2023).
- Hemberg, Erik et al., *Linking Threat Tactics, Techniques, and Patterns with Defensive Weaknesses, Vulnerabilities and Affected Platform Configurations for Cyber Hunting*, 2021, arXiv: 2010.00533 [cs.CR].
- IBM, *Cost of a Data Breach Report 2023*, URL: <https://www.ibm.com/reports/data-breach> (visited on 07/26/2024).
- Inc, Ansible, *Ansible*, 2012, URL: <https://www.ansible.com/>.
- Ingols, Kyle, Richard Lippmann, and Keith Piwowarski, « Practical Attack Graph Generation for Network Defense », in: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006, pp. 121–130, DOI: 10.1109/ACSAC.2006.39.
- Janisch, Jaromír, Tomáš Pevný, and Viliam Lisý, *NASimEmu: Network Attack Simulator and Emulator for Training Agents Generalizing to Novel Scenarios*, 2023, arXiv: 2305.17246 [cs.CR].
- Jaromír, *NASimEmu: Network Attack Simulator and Emulator*, 2023, URL: <https://github.com/jaromiru/NASimEmu/>.
- Kidd, Chrissy, *Cyber Kill Chains Explained: Phases, Pros/Cons and Security Tactics*, 2022, URL: https://www.splunk.com/en_us/blog/learn/cyber-kill-chains.html (visited on 03/04/2024).
- Kuppa, Aditya, Lamine Aouad, and Nhien-An Le-Khac, « Linking CVE's to MITRE ATT&CK Techniques », in: *Proceedings of the 16th International Conference on Availability, Reliability and Security*, ARES '21, Vienna, Austria: Association for Computing Machinery, 2021, ISBN: 9781450390514, DOI: 10.1145/3465481.3465758, URL: <https://doi.org/10.1145/3465481.3465758>.
- Landauer, Max et al., « Have it Your Way: Generating Customized Log Datasets With a Model-Driven Simulation Testbed », in: *IEEE Transactions on Reliability* 70.1 (2021), pp. 402–415, DOI: 10.1109/TR.2020.3031317.
- Landauer, Max et al., « Maintainable Log Datasets for Evaluation of Intrusion Detection Systems », in: *IEEE Transactions on Dependable and Secure Computing* 20.4 (2023),

-
- pp. 3466–3482, DOI: 10.1109/tdsc.2022.3201582, URL: <https://doi.org/10.1109/2Ftdsc.2022.3201582>.
- Limited, Puppet Labs, *Puppet*, 2005, URL: <https://www.puppet.com/> (visited on 10/23/2023).
- lockheedmartin, *The Cyber Kill Chain*, 2011, URL: <https://honeyscore.shodan.io/> (visited on 08/30/2023).
- Lyon, Gordon, *Nmap*, 1997, URL: <https://nmap.org/> (visited on 08/30/2023).
- McCarthy, Jay, *Datalog*, 1977, URL: <https://docs.racket-lang.org/datalog/index.html> (visited on 11/22/2023).
- Mensah, Pernelle, « Generation and Dynamic Update of Attack Graphs in Cloud Providers Infrastructures », Theses, CentraleSupélec, June 2019, URL: <https://inria.hal.science/tel-02416305>.
- MITRE, *APT3 Adversary Emulation Plan*, 2018, URL: http://attack.mitre.org/docs/APT3_Adversary_Emulation_Plan.pdf (visited on 11/20/2023).
- *ATT&CK v13 Enters the Room: Pseudocode, Swifter Search, and Mobile Data Sources*, 2023, URL: <https://medium.com/mitre-attack/attack-v13-enters-the-room-5cef174c32ff> (visited on 11/20/2023).
- *Attack Flow*, 2022, URL: <https://center-for-threat-informed-defense.github.io/attack-flow/overview/>.
- *CVE Program Mission*, URL: <https://www.cve.org/> (visited on 04/20/2024).
- *Enterprise Matrix*, 2013, URL: <https://attack.mitre.org/> (visited on 10/24/2023).
- Ngrok, URL: <https://ngrok.com/> (visited on 07/05/2024).
- NIST, *CVE-2016-5195 Detail*, 2016, URL: <https://nvd.nist.gov/vuln/detail/cve-2016-5195> (visited on 04/20/2024).
- *CVE-2017-0005 Detail*, 2017, URL: <https://nvd.nist.gov/vuln/detail/CVE-2017-0005> (visited on 04/20/2024).
- *CVE-2019-0708 Detail*, URL: <https://nvd.nist.gov/vuln/detail/cve-2019-0708> (visited on 07/26/2024).
- *CVE-2023-0386 Detail*, 2023, URL: <https://nvd.nist.gov/vuln/detail/CVE-2023-0386> (visited on 04/20/2024).
- NVD, *CVE-2019-14287 Detail*, 2019, URL: <https://nvd.nist.gov/vuln/detail/CVE-2019-14287> (visited on 08/30/2023).
- *CVE-2021-4034 Detail*, 2022, URL: <https://nvd.nist.gov/vuln/detail/CVE-2021-4034> (visited on 08/30/2023).

-
- *CVE-2021-40444 Detail*, 201, URL: <https://nvd.nist.gov/vuln/detail/CVE-2021-40444> (visited on 07/26/2024).
 - *Search Common Platform Enumerations*, 2009, URL: <https://nvd.nist.gov/products/cpe/search> (visited on 04/30/2024).
 - Oracle, *Introduction to Networking Modes*, URL: <https://www.virtualbox.org/manual/ch06.html> (visited on 05/30/2024).
 - *VirtualBox*, 2007, URL: <https://www.virtualbox.org/> (visited on 10/02/2023).
 - Orsini, Helene et al., « AdvCat: Domain-Agnostic Robustness Assessment for Cybersecurity-Critical Applications with Categorical Inputs », in: *BigData 2022 - IEEE International Conference on Big Data*, Osaka, Japan: IEEE, Dec. 2022, pp. 1–13, URL: <https://inria.hal.science/hal-03893496>.
 - Ou, Xinming, Wayne F. Boyer, and Miles A. McQueen, « A Scalable Approach to Attack Graph Generation », in: *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, Alexandria, Virginia, USA: Association for Computing Machinery, 2006, pp. 336–345, ISBN: 1595935185, DOI: 10.1145/1180405.1180446, URL: <https://doi.org/10.1145/1180405.1180446>.
 - Outkin, Alexander V. et al., « Defender Policy Evaluation and Resource Allocation With MITRE ATT&CK Evaluations Data », in: *IEEE Transactions on Dependable and Secure Computing* 20.3 (2023), pp. 1909–1926, DOI: 10.1109/TDSC.2022.3165624.
 - OVH, *OVHCloud*, 1999, URL: <https://www.ovhcloud.com/fr/> (visited on 08/02/2024).
 - Paganini, Pierluigi, *APT3 Operation Double Tap is targeting recently disclosed Windows vulnerabilities*, 2014, URL: <https://securityaffairs.com/30528/cyber-crime/apt3-operation-double-tap.html> (visited on 09/06/2023).
 - Pham, Cuong et al., « CyRIS: a cyber range instantiation system for facilitating security training », in: Dec. 2016, pp. 251–258, DOI: 10.1145/3011077.3011087.
 - Phillips, Cynthia and Laura Painton Swiler, « A graph-based system for network-vulnerability analysis », in: *Proceedings of the 1998 Workshop on New Security Paradigms*, NSPW '98, Charlottesville, Virginia, USA: Association for Computing Machinery, 1998, pp. 71–79, ISBN: 1581131682, DOI: 10.1145/310889.310919, URL: <https://doi.org/10.1145/310889.310919>.
 - Pierre-Victor, Besson et al., « URSID: Automatically Refining a Single Attack Scenario into Multiple Cyber Range Architectures », in: *Foundations and Practice of Security - FPS 2023*, vol. 14551, 2024, DOI: 10.1007/978-3-031-57537.

-
- Poisson, Manuel et al., « Unveiling stealth attack paths in Windows Environments using AWARE », in: *CSNet 2023 - 7th Cyber Security in Networking Conference*, IEEE ComSoc, Montreal, Canada, Oct. 2023, pp. 1–7, URL: <https://inria.hal.science/hal-04163780>.
- pySigma Elasticsearch Backend*, URL: <https://github.com/SigmaHQ/pySigma-backend-elasticsearch> (visited on 07/10/2024).
- pySigma Splunk Backend*, URL: <https://github.com/SigmaHQ/pySigma-backend-splunk> (visited on 07/10/2024).
- Russo, Enrico, Gabriele Costa, and Alessandro Armando, « Scenario Design and Validation for Next Generation Cyber Ranges », in: *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, 2018, pp. 1–4, DOI: 10.1109/NCA.2018.8548324.
- Sanchez, Alexandre, *Casino Limit Challenge*, 2024, URL: <https://gitlab.inria.fr/pirat-public/casinolimit> (visited on 07/23/2024).
- Schreuders, Z. Cliffe et al., « Security Scenario Generator (SecGen): A Framework for Generating Randomly Vulnerable Rich-scenario VMs for Learning Computer Security and Hosting CTF Events », in: *2017 USENIX Workshop on Advances in Security Education (ASE 17)*, Vancouver, BC: USENIX Association, Aug. 2017, URL: <https://www.usenix.org/conference/ase17/workshop-program/presentation/schreuders>.
- Schwartz, Jonathon, *Network Attack Simulator*, 2018, URL: <https://github.com/Jjschwartz/NetworkAttackSimulator>.
- Sharma, Yashovardhan, Simon Birnbach, and Ivan Martinovic, *RADAR: A TTP-based Extensible, Explainable, and Effective System for Network Traffic Analysis and Malware Detection*, 2023, arXiv: 2212.03793 [cs.CR].
- Sigma - Generic Signature Format for SIEM Systems*, URL: <https://github.com/SigmaHQ/sigma> (visited on 07/10/2024).
- SIGMA detection rules*, URL: <https://github.com/mdecrevoisier/SIGMA-detection-rules> (visited on 07/10/2024).
- SigmaHQ, URL: <https://github.com/SigmaHQ/sigma> (visited on 07/05/2024).
- Sophos, *The State of Ransomware 2024*, URL: <https://assets.sophos.com/X24WTUEQ/at/9brgj5n44hqvgsp5f5bqcps/sophos-state-of-ransomware-2024-wp.pdf> (visited on 07/26/2024).
- Suricata, URL: <https://suricata.io/> (visited on 07/05/2024).

-
- Talon, Natan et al., « SCWAD: Automated Pentesting of Web Applications », *in: Proceedings of the 21st International Conference on Security and Cryptography - Volume 1: SECRYPT*, INSTICC, SciTePress, 2024, pp. 424–433, ISBN: 978-989-758-709-2, DOI: 10.5220/0012721000003767.
- Things, Internal All The, *Reverse Shell Cheat Sheet*, URL: <https://swisskyrepo.github.io/InternalAllTheThings/cheatsheets/shell-reverse-cheatsheet/>.
- Uetz, Rafael et al., « Reproducible and Adaptable Log Data Generation for Sound Cybersecurity Experiments », *in: Annual Computer Security Applications Conference*, ACM, 2021, DOI: 10.1145/3485832.3488020, URL: <https://doi.org/10.1145/2F3485832.3488020>.
- University, Carnegie Mellon, URL: <https://github.com/cmu-sei/GHOSTS> (visited on 07/05/2024).
- Venkatesan, Sridhar et al., « VulnerVAN: A Vulnerable Network Generation Tool », *in: MILCOM 2019 - 2019 IEEE Military Communications Conference (MILCOM)*, 2019, pp. 1–6, DOI: 10.1109/MILCOM47813.2019.9021013.
- Yamin, Muhammad Mudassar and Basel Katt, « Modeling and executing cyber security exercise scenarios in cyber ranges », *in: Computers & Security* 116 (2022), p. 102635, ISSN: 0167-4048, DOI: <https://doi.org/10.1016/j.cose.2022.102635>, URL: <https://www.sciencedirect.com/science/article/pii/S0167404822000347>.
- Zhong, Shangqin, Danfeng Yan, and Chen Liu, « Automatic Generation of Host-Based Network Attack Graph », *in: 2009 WRI World Congress on Computer Science and Information Engineering*, vol. 1, 2009, pp. 93–98, DOI: 10.1109/CSIE.2009.102.
- Zircolite - Standalone SIGMA-based detection tool, URL: <https://github.com/wagga40/Zircolite> (visited on 07/10/2024).

Titre : Génération automatisée d'architectures vulnérables variables.

Mot clés : Scénarios d'attaques, Architecture vulnérables, Cyber ranges

Résumé : La capacité à générer des architectures vulnérables est très utile du point de vue de la défense cyber, afin d'aider à l'entraînement d'experts en cyber sécurité ou de générer des jeux de données d'attaquants. Nous associons ici une architecture vulnérable avec leur scénario d'attaque correspondant, i.e la séquence d'actions qu'un attaquant doit performer afin d'accomplir ses objectifs sur celle ci. La question de la modélisation de scénarios est donc liée à celle du déploiement d'architectures vulnérables. Cette modélisation se doit d'être assez précise afin de décrire ou pouvoir inférer la configuration de l'architecture correspondante, mais trop d'exhaustivité dans cette représentation limite la réusabilité de ces descriptions. Nous proposons un nou-

veau paradigme de déploiement d'architecture vulnérable qui vise à améliorer le processus sur ces aspects comparé à l'état de l'art. Nous décrivons des scénarios d'attaque avec un formalisme haut niveau, qui combine les notions de graphes d'attaque et de la matrice MITRE ATT&CK. Nous sommes ensuite capable de raffiner automatiquement cette description en plusieurs descriptions bas niveau, qui diffèrent sur leurs détails d'implémentation mais correspondent toutes au scénario d'attaque original. Nous deployons enfin ces descriptions en tant que réseaux vulnérables de machines virtuelles. URSID, l'outil résultant du travail de cette thèse, est disponible en open source et a été utilisé avec succès pour conduire deux expériences à but éducatif.

Title: Automated generation of variable vulnerable architectures.

Keywords: Attack scenarios, Vulnerable architectures, Cyber ranges

Abstract:

The ability to generate vulnerable architectures is very valuable for cyber defense purposes, as it helps with the training of cyber security experts or to generate attacker datasets. We associate vulnerable architectures with the sequence of actions attackers are expected to perform in order to accomplish their goals in said architecture, which we refer to as an attack scenario. The question of attack scenariomodelization is thus at the center of vulnerable architecture geneation. This description should contain or infer all the information necessary to deploy the corresponding architecture, but being too exhaustive on this representation limits the reusability and ease of use of a

given description. We thus offer a new vulnerable architecture deployment paradigm aimed at improving on those aspects compared to the state of the art. We describe attack scenarios with a high level formalism, which makes use of attack graphs and the MITRE ATT&CK matrix. We are then able to automatically refine this description into several low level variations, which differ on specific implementation details but follow the same original attack scenario. We are then able to deploy these descriptions as vulnerable virtual machine networks. URSID, the tool resulting from this work, is available as open source and has been used to successfully conduct two experiments with educational values.